7. REST

7.1 Learning Objectives

The learning objective in this chapter is the fundamental concepts and principles in the **Representational State Transfer (REST)** architectural style. This style of programming distributed systems has a number of attractive properties when it comes to designing and implementation systems that must be scalable, that is, support many clients making requests at the same time.

7.2 The Demise of Broker Architectures

If you look broadly at how people architect distributed systems today, you will notice that REST based architectures is used and dicussed a lot, while Broker based architectures are not. Then, you may wonder why learn the *Broker* pattern at all?

My personal view (and I stress that it is an opinion, not something I can claim based upon scientific studies) is that *Broker* is a strong pattern to base distributed systems upon, *if you do it correctly*. And, in the heydays of Broker in the 1990'ies, most developers got it wrong, the resulting systems suffered from a lot of issues with performance, maintainability, and availability, and the natural conclusion was that the pattern was broken.

I attribute this misunderstanding from the fact that early proponents of the *Broker* pattern stated that *tranparency* was an important feature, in the sense that you were now able to code a method call `obj.doSomething()` and it was *transparent* whether the `obj` was a local or a remote object. Being able to program distributed systems using the object oriented paradigm was a major improvement compared to programming in terms of low-level `send()` and `receive()` calls, as argued in Chapter Basic Concepts - however - the ideal of transparency is not so! As argued several times already, a remote object is architecturally highly different from a local object. Thus, early broker based systems became slow, insecure, and volatile, because architects and programmers were not sufficiently focused on keeping a strong borderline between remote and local objects: server objects became highly coupled to client side objects, suddenly the server invokes methods on thousands of client side objects, distributed calls were not sufficiently guarded against network or server failures, etc.

Another issue leading to abandoning *Broker*, again in my opinion, was that it was supported by frameworks such as Corba, Java RMI, and .NET remoting. While lowering the implementation burden, which is good, it also tied developers into fixed decisions regarding marshalling formats and IPC protocols, which leaves less room for architects to choose the right technique for the given challenges.

So, the issue was not really the *Broker* pattern per se, it was the idea of transparency and fixed architectural decisions that lead to ill designed systems. And while these systems showed weaknesses, the HTTP protocol based world wide web showed another and seemingly much more robust approach to architecting distributed systems. No wonder, software developers quickly moved there instead. *Representational State Transfer (REST)*, described in this chapter, is based upon the ideas of the world wide web and defines an architectural pattern that utilizes the key ideas of HTTP and WWW in a programming context. And one key aspect of REST is the clear separation of responsiblities of the client side and the server side respectively, that is, the exact opposite of *transparency*. And, as I have done with my *Broker* pattern, REST is a clear client-server system: servers do not call clients.

A second issue was that an object by nature stores *state*: instance variables hold values that can be set and retrieved, defining the state of that object. However, it also follows that once a server creates an object, *it only exists on that particular server*. This has two problematic implications in a distributed setting: Firstly, if the server crashes or has to be stopped (for instance, to install a security update in the operation system), all state vanishes which of course is not acceptable. Secondly, if the number of clients being served by the server grows, the server becomes a bottleneck until the point where it can no longer keep up and becomes overloaded. An overloaded server is at best unbearably slow and at worst simply crashed. To avoid this, you have to *scale out*, and introduce more server machines and load balancing, but then object state cannot be tied to a particular server - any server in the set have to be able to recreate any object's state. Your servers have to be *stateless*. My TeleMed system's server was stateless as any uploaded measurement was immediately stored in the database, and retrievals were not based upon objects held in the server's memory. So, an important aspect of creating server programs is to make it stateless, a key idea that was not considered in the original formulations of the *Broker* pattern.

Thus, my final note on *Broker* is that it *is* a strong architectural pattern for architecting distributed systems *if and only if* you as an architect keeps a clear separation of which objects are local and which are remote, and ensure that your server is stateless. If you do that, then *Broker* has a programming model that in all respects are much nicer than the REST model, as you will experience in this chapter.

7.3 Representational State Transfer (REST)

Fielding and Taylor presented an architectural pattern, *Representational State Transfer (REST)* in the paper *Principled Design of the Modern Web Architecture*[1], for the purpose of handling *internet-scale* systems.

Their basic goal was to *Keep the scalable hypermedia properties of the World Wide Web*. This leads to five central principles of REST:

- The central concept is the *resource* which is defined as any information that can be named.
- A resource is identified by a *resource identifier*.
- You transfer a *representation of data* in a format matching one of the standard media types.
- All interactions are *stateless*, i.e. every request must contain all the information necessary for processing it.
- Interactions are between *clients* and *servers*, the essential difference between the two is that a client initiates communication by making a request, whereas a server listens to connections and responds to requests in order to supply access to its services.

Most of these aspect have already been treated in the previous chapter on HTTP, and I argued for the advantages of clear separation between client and server objects, and stateless interactions as vital for large-scale systems in the introduction.

The REST principles by themselves are pretty abstract so I will make them more concrete in the following sections.

7.4 Richardson's model for Levels in REST

In the book, *REST in Practice*[2], the authors discusses distributed systems that uses the web and properties of the HTTP protocol, and present a model developed by Leonard Richardson to explain their maturity[3]. Here "maturity" is in terms of how many features and principles from the REST style they use. Richardson identifies three levels and number them accordingly:

- Level 0: URI Tunneling

[1] Roy Fielding, and Richard N. Taylor: "Principled design of the modern Web architecture", Proceedings of the 22nd international conference on Software engineering, Ireland, 2000.

[2] Jim Webber, Savas Parastatidis, and Ian Robinson: "REST in Practice: Hypermedia and Systems Architecture", O'Reilly Media, 2010.

[3] If you search for the Richardson model on WWW, you may find descriptions in which four levels are mentioned. However, I follow the presentation given in the "Rest in Practice" book.

- Level 1: HTTP Verbs
- Level 2: Hypermedia

In *Level 0: URI Tunneling* systems, HTTP is simply used as a well supported IPC layer, just as I implemented the IPC layer of the Broker pattern using HTTP and used the Spark and UniRest libraries for implementation. None of the properties of HTTP is used at this level, it is only the IPC. Recall, that our TeleMed system only used the POST verb, and all method calls were made to a resource named "/bp" - which thus was not really a resource per se, just a way to identify the path the server and client had to communicate by.

Another and more prominent example of URI Tunneling is **SOAP**, originally *Simple Object Access Protocol*, which gained a lot of attention around year 2000. SOAP is basically the *Broker* on top of HTTP, and a lot of tools to generate **ClientProxies** and **Invokers** code automatically.

At *Level 1: HTTP Verbs*, systems start obeying the HTTP requirements: You design your system around resources with a resource identifier, and you use the HTTP verbs to create, read, update, and delete these resources. My *PasteBin* example in the previous chapter was a simple Level 1 system: Each POST created a new resource and told the client the resource identity of it (like "/bin/102"), which allowed the client to read it. Adding update and delete features to the PasteBin client is a simple exercise.

Level 1 systems can handle a lot of simple systems that match the CRUD template. In Section Level 1 REST: TeleMed below, I will showcase how to (more or less) implement TeleMed using level 1 REST.

The final level is *Level 2: Hypermedia* which uses hypermedia links to model application state changes, which departs radically from traditional object-oriented way of handling state changes. I will outline the concepts in Section Level 2 REST: GameLobby, and demonstrate it on our GameLobby system.

7.5 The Architectural Style

REST is an *architectural style/architectural pattern*, that is, a certain way of organizing your software and design. As such it leads to other structures and design decisions than those in an object-oriented style.

In object-orientation, you have objects that send messages to each other: one object invokes a method on another. Methods often follow the *Command Query Separation (CQS)* principle, that is, some methods are *accessors/queries* that retrieve state information, while others are *mutators/commands* that change state in the object. The classic example is the Account object with `getBalance()` accessor and `withdraw(amount)` mutator. This style allows state/data to be encapsulated with all sorts of different methods for manipulation.

REST is in contrast a *data-centric style.* In essence, it is based upon the named resource, the data, and just supports the four basic operations of Create, Read, Update, and Delete on the resource, as outlined in the previous chapter. Comparing to objects, REST is just the object's fields/instance variables, and there are only these four fixed methods available. No other methods can be defined.

Going back to our PasteBin application, you saw the REST style in practice. There was just a single root resource, the named information **Bin**, that represents a clipboard entry. A resource is identified by the path `/bin/`. By sending this resource a POST message with a JSON object, I told that I wanted a new sub-object bin created with the given data, which in turn created such a named resource with a name, `/bin/100`. This new resource I can then update (PUT), or read (GET), or delete (DELETE).

This is all well for designing systems that fit a data-centric, database flavor, style. Indeed *information systems: organized systems for the collection, organization, storage and communication of information* which abound on the WWW fits the style well. As examples, consider internet shops (Create a shopping basket, Update its contents with items to buy, Read its contents so I can see it, Delete it once I have paid for the items), or social media (Create a profile, Read my own profile and that of my friends, Update my profile with images and text.)

However, many domains do not fit well with this data-centric design. Generally, systems that have complex state changes and in which many objects are affected by a single operation do no fit well. We will return to how to design such systems, but first, let us return to our TeleMed system, and see how a REST design may look.

7.6 Level 1 REST: TeleMed

We have already implemented the TeleMed system using the Broker pattern, and in the last chapter I used HTTP to implement the IPC layer. It was not REST, though; I only used POST messages on a single resource named /bp to handle all TeleMed methods.

TeleMed actually fits the *Level 1 REST style* well: We create TeleObservations of blood pressure and the patient as well as physicians read them.

How can our TeleMed system be designed using the REST style? The central REST concept is the *resource* which is named information. A specific measured blood pressure measurement for a specific patient fits this requirement well. So we need to identify it using a URI, remember the URI schema. My suggestion is to use a path that encodes a given patient and given instance of a measurement, something akin to

```
/bloodpressure/251248-1234/id73564823
```

Breaking down this URI path, it encodes the two pieces of information, using the patient id (251248-1234), and some computer generated id (id73564823) for that specific measurement. Another potential path may be even more readable, if we use the time as the last part of the path, like

```
/bloodpressure/251248-1234/2020-05-14-14-48-53
```

to identify the measurement made May 14th 2020 at 14.48.53 in the 24hour clock.

The above schema also follows the rule that an URI should name *things*, and not actions. Paths consist of *nouns*, never *verbs*, because paths identify resources which are "things".

Now that I have a resource, so I need to access it. In REST you do not manipulate data directly, instead you manipulate a *representation of data* in a well known media type, such as XML and JSON. A direct manipulation example could be issuing SQL queries to a SQL server, or invoking XDS methods on an XDS infrastructure. REST enforces a *uniform interface* instead, requiring you to manipulate data in your representation of choice from the limited set of standard media types.

So, to request a Read (GET) on this particular resource on the server assigned for such measurements, you would set a HTTP request stating that you need a JSON encoded representation of the measurement

```
GET /bloodpressure/251248-1234/id73564823 HTTP/1.1
Accept: application/json
```

which should return something like

```
{
  patientId:  "251248-1234",
  systolic:   128.0,
  diastolic:  76.0,
  time: "20180514T144853Z"
}
```

This *GET on specific URI* just returns a single measurement. However, **Story 2: Review blood pressure** requires that all measurements for the last week can be displayed.

A de-facto way to support this is just to GET on the patient specific URI without the measurement specific part, that is on

```
/bloodpressure/251248-1234
```

This path may be coded to always return exactly the last week's measurements as a JSON array; or I may code the server to look for URI query parameters like

```
/bloodpressure/251248-1234?interval="week"
```

which can then be used to filter the returned data.

The other important user story is **Story 1: Upload a blood pressure measurement.** As this story is about uploading data to the server, which must store it, it is essentially a Create / POST. So the REST way is to let the client perform a POST on the URI for the particular patient, with a JSON payload with the measured values

```
POST /bloodpressure/251248-1234 HTTP/1.1
Content-Type: application/json

{
  patientId:  "251248-1234",
  systolic:   144.0,
  diastolic:  87.0,
  time: "20180515T094804Z"
}
```

and the server must then store the measurement, define a new URI that represents it, and return it in the *Location* for the newly created resource, similar to

```
HTTP/1.1 201 Created
Date: Mon, 07 May 2018 12:16:51 GMT
Content-Type: application/json
Location: http://telemed.baerbak.com/bloodpressure/251248-1234/id73564827

{
  patientId:  "251248-1234",
  systolic:   144.0,
  diastolic:  87.0,
  time: "20180515T094804Z"
}
```

7.7 Documenting REST API

In object-oriented designs we use interfaces to document how to interact with our roles. I need a similar thing for my REST designs.

One important initiative for documenting REST interactions is the OpenAPI Initiative[4] which is a JSON format supported by tools such as editors and code generators. The downside of OpenAPI is the detail required to specify an interface.

Therefore, I instead propose a simple text format that show/exemplify the HTTP verbs, payloads, and responses. It is rudimentary and less strict, but much less verbose.

The text format is simply divided into section, one for each REST operation, and just provides examples for data payloads, header key-value pairs, paths, etc.

Let us jump right in, using a TeleMed REST API as example.

```
TeleMed
========

Create new tele observation
----------

POST /bloodpressure/{patient-id}

{
  systolic:   128.0,
  diastolic:  76.0,
  time: "20180514T144853Z"
}

Response
  Status: 201 Created
  Location: /bloodpressure/{patient-id}/{measurement-id}

Get single measurement
-------

GET /bloodpressure/{patient-id}/{measurement-id}
  (none)

Response
```

[4] https://swagger.io/resources/open-api/

```
  Status: 200 OK
  { systolic:   128.0,
    diastolic:  76.0,
    time: "20180514T144853Z" }

  Status: 404 Not Found
  (none)

Get last week's measurement
-------

GET /bloodpressure/{patient-id}
  (none)

Response
  Status: 200 OK
  [
    { systolic:   124.1,
      diastolic:  77.0,
      time: "20180514T073353Z" },
    { systolic:   128.8,
      diastolic:  76.2,
      time: "20180514T144853Z" },
    { systolic:   132.5,
      diastolic:  74.2,
      time: "20180514T194853Z" }, ... ]

  Status: 404 Not Found
  (none)
```

In this format each "method" starts with a headline, like "Get single measurement" etc., and is then followed by the request and reply format. Regarding the request I write the HTTP Verb and the URI path, and potential message body/payload. I use curly braces for parameter values so for instance `{patient-id}` must be replaced by some real id of a patient. The response is outlined after the request section, and may have multiple sections, each showing a particular HTTP status codes and examples of the format of the returned payloads; empty lines separate a list of possible return status codes, like "200 OK" or "404 Not Found" in the examples above.

7.8 Continued REST Design for TeleMed

For the sake of demonstrating the remaining HTTP verbs, I have enhanced the TeleMed interface a bit so a patient can update and delete specific

measurements. (It should be noted that this is not proper actions in a medical domain – you do not delete information in medical journals, rather you add new entries that mark previous ones as incorrect and add the correct values.)

The TeleMed interface is then augmented with two additional methods

```
1  public interface TeleMed {
2
3    // methods 'processAndStore', and 'getObservationsFor' not shown
4
5    /**
6     * Return the tele observation with the assigned ID
7     *
8     * @param uniqueId
9     *          the unique id of the tele observation
10    * @return the tele observation or null in case it is not present
11    * @throws IPCException in case of any IPC problems
12    */
13   TeleObservation getObservation(String uniqueId);
14
15   /**
16    * Correct an existing observation, note that the time stamp
17    * changes are ignored
18    *
19    * @param uniqueId
20    *          id of the tele observation
21    * @param to
22    *          the new values to overwrite with
23    * @return true in case the correction was successful
24    * @throws IPCException in case of any IPC problems
25    */
26   boolean correct(String uniqueId, TeleObservation to);
27
28   /**
29    * Delete an observation
30    *
31    * @param uniqueId
32    *          the id of the tele observation to delete
33    * @return true if the observation was found and deleted
34    * @throws IPCException in case of any IPC problems
35    */
36   boolean delete(String uniqueId);
37 }
```

Note that these three additional methods match the GET on single resource, UPDATE, and DELETE operations respectively; and that the parameters are

rather REST inspired by operating through a 'uniqueId' string that is thus similar to a (part of a) resource identifier.

7.9 Implementing REST based TeleMed

While I have argued that Broker-based designs and REST are two different architectural styles, there are of course quite some similarities, as they both provide solutions to the basic challenges of remote communication:

- The *Request-Reply protocol* is central in both styles. In Broker, we may have to code it directly in the RequestHandlers if we use sockets, while the HTTP already implements it.
- The need for *Marshalling* of data contents. In Broker, we again coded it in the Requestor/Invoker pair, while HTTP relies on media types.
- The need for *Name services*. In Broker, we used DNS systems to get the IP address of the server, while we used an internal implementation for getting a servant object associated with a given object id. REST, on the other hand, encodes everything into the URI: both the server identity as well as the resource identity.
- The *Proxy* is only an issue for object-oriented designs.

Looking at HTTP and thus REST I will argue that it basically *merges the layers* and does a lot of *hard coupling*. If we take the TeleMed POST message above

```
1 POST /bloodpressure/251248-1234 HTTP/1.1
2 Content-Type: application/json
```

it defines the IPC layer (HTTP over TCP/IP), it binds the marshalling layer (JSON and the HTTP text based message format), and actually also binds the domain layer in the respect that the URI's parameter list define a resource named "bloodpressure" and social security number as the unique identifier.

Thus, the client side TeleMed object, serves *all* three client-side roles of the Broker pattern: it is the **ClientProxy** as it implements the TeleMed interface, it is the **Requestor** as it does the marshalling, and it is also the **ClientRequestHandler** as it performs the IPC calls. Thus, the `processAndStore()` method (in a class I could name `TeleMedClientProxyRequestorClientRequestHandler`, but no, I will not) will become something like

(Fragment in *telemed-rest* project: TeleMedRESTProxy.java)

```
@Override
public String processAndStore(TeleObservation teleObs) {
    // Marshalling
    String payload = gson.toJson(teleObs);
    HttpResponse<JsonNode> jsonResponse = null;

    // IPC
    String path = "/bloodpressure/" + teleObs.getPatientId() + "/";
    try {
        jsonResponse = Unirest.post(serverLocation + path).
                header("Accept", MimeMediaType.APPLICATION_JSON).
                header("Content-type", MimeMediaType.APPLICATION_JSON).
                body(payload).asJson();
    } catch (UnirestException e) {
        throw new IPCException("UniRest POST failed for 'processAndStore'", e);
    }

    // Extract the id of the measurement from the Location header
    String location = jsonResponse.getHeaders().getFirst("Location");

    [extract teleObsID from the location header]

    return teleObsID;
}
```

Note that this method on the client side *is* the ClientProxy, *does* the marshalling/demarshalling and *does* perform all IPC.

On the server side, the layers are also merged in a single implementation - it *is* the **ServerRequestHandler** as it receives network messages, and is the **Invoker** as it is does the demarshalling and marshalling, and the invocation of the **Servant** method.

(Fragment in *telemed-rest* project: RESTServerRequestHandlerInvoker.java)

```
// IPC
String storeRoute = "/bloodpressure/:patientId/";
post(storeRoute, (req, res) -> {
  String patientId = req.params(":patientId");
  String body = req.body();

  // Demarshall parameters into a JsonArray
  TeleObservation teleObs = gson.fromJson(body, TeleObservation.class);

  // Invoker
  String id = teleMed.processAndStore(teleObs);

```

```
13    // Normally: 201 Created
14    res.status(HttpServletResponse.SC_CREATED);
15    res.type(MimeMediaType.APPLICATION_JSON);
16
17    // Location = URL of created resource
18    res.header("Location",
19        req.host() + "/" + Constants.BLOODPRESSURE_PATH + id);
20
21    // Marshalling return value
22    return gson.toJson(teleObs);
23  });
```

The `post` method binds the URI /bloodpressure/{patientId} so all incoming requests are routed to the lambda function enclosed. And this lambda function in turn does demarshalling and up call to the Servant, and replies to the client. All layers are thus merged.

Compared to the Broker, REST thus have fewer hot spots and injection points, and thus no separation of concerns. This has the negative consequence that testability is lowered, as I have no way of injecting a test double in the place of the IPC layer. This is a major liability, and I will return later in this chapter to techniques to mitigate it.

You can find the detailed code in folder *telemed-rest* at FRDS.Broker Library[5]. Note that this code differs somewhat from the design provided here in the book. As the patient ID is also present in the exchanged JSON documents, the patient id part of the URI is not implemented in the source code.

Note also that the code base does not contain any automated JUnit tests, it is pure manual testing.

7.10 Level 2 REST: GameLobby

As I have described, REST fits a CRUD schema well, but not everything in computing is just creating, reading, updating, and deleting a single piece of information. Many applications must perform complex state changes involving a number of objects, something the OO style is well suited for.

To model that, REST relies on the *hypermedia concept* which is actually well known from ordinary browsing web pages: You read a web page and it contains multiple hypermedia links that you can follow by clicking. In this sense, each web page contains a set of links that define *natural state changes* in the information the user receives. In the same manner, Level 2 REST defines state changes by returning links as part of the returned resources, each link in itself a resource that can be acted upon using the POST, GET, etc.

[5] www.bitbucket.com/henrikbaerbak/broker

To illustrate, consider a catalog web page for a small web shop. The page presents a set of items that can be purchased and next to each item is a link named ‘add this item to shopping basket’. Clicking on such a link must naturally add the item to the basket, so the next time the user visits the shopping basket web page, the item appears on the list. Thus, besides a “/catalog” resource, the system also maintains a “/shoppingbasket/654321” (the shopping basket for user with id 654321) resource; and the ‘add item to basket’ link will refer to this resource, and clicking it will make the client issue an UPDATE request to it.

This way of modeling state changes is also known by the acronym **HATEOAS**, *Hypermedia As Engine Of Application State*.

GameLobby

One example that I will return to is the GameLobby introduced previously. The **Story 1: Creating a remote game** match a create operation thus a POST on a root path/resource that I name /lobby. When posting I must provide two parameters, namely my player name and the level of the game I want to play.

So the equivalent of the call:

```
1  FutureGame player1Future = lobbyProxy.createGame("Pedersen", 0);
```

in the Broker based version of the code (See Walk-through of a Solution) becomes something along the lines of the following REST call:

```
1  GameLobby
2  ========
3
4  Create Remote Game
5  ----------
6
7  POST /lobby/
8
9  {
10   playerOne: "Pedersen",
11   level: 0
12 }
13
14 Response
15   Status: 201 Created
16   Location: /lobby/{future-game-id}
17
18   {
19     playerOne: "Pedersen",
```

```
    playerTwo: null,
    level: 0,
    available: false,
    next: null
  }
```

In the Broker variant, a FutureGame instance is returned which can be queried for a *joinToken*. In my REST create operation, a new resource is created and its location also returned: `/lobby/{future-game-id}`; which serves nicely as our joinToken. Thus, Pedersen will simply tell Findus to join his game using that resource path. The returned resource is a JSON object embodying the future game resources, with the name of the players, the availability state, and a empty links section ("next: null") which I will describe below. Note that our initial POST's body only included values for keys that was meaningful for the client to define: first player's name and the game's level. The other key-value pairs were not defined. POST bodies are allowed to only provide partial resource definitions.

It would be obvious to add a read operation on the resource, to allow Pedersen to test if Findus has indeed joined the game. Reading translates to a GET operation.

```
Read future game status
----------

GET /lobby/{future-game-id}

Response
  Status: 200 OK

  {
      playerOne: "Pedersen",
      playerTwo: null,
      level: 0,
      available: false,
      next: null
  }

  Status: 404 Not Found
  (none)
```

Story 2: Joining an existing game translates to updating the future game's resource with the missing information (in our case, just the second player's name). Updating is normally a PUT operation, **but** PUT's body is defined to be the *full representation* of the resource. As Findus only wants to make a partial update, supplying his own name as player two, we cannot use the PUT verb, but have to use the POST verb.

There seems to be different opinions on how to update resources with partial information. Just using PUT seems obvious, but most authors agree that the strict semantics of *full representation in PUT body* removes this possibility. And a general recommendation is therefore to use POST[6]. The HTTP verb, PATCH, has been added to handle partial updates, but is complex and not widely used. The complexity is because the RFC 5789 specification states that *With PATCH, however, the enclosed entity contains a set of instructions describing how a resource currently residing on the origin server should be modified to produce a new version.* So, you do not simply send a partial JSON object, but must use a special patch format that both client and server agrees upon. For the sake of simplicity, I will stick to POST.

```
Join A Game
-----------

POST /lobby/{future-game-id}

{
  playerTwo: "Findus"
}

Response
  Status: 200 OK

  {
      playerOne: "Pedersen",
      playerTwo: "Findus",
      level: 0,
      available: true,
      next: "/lobby/game/{game-id}"
  }

  Status: 404 Not Found
  (none)
```

There are several things to note about this proposed POST operation

- The outcome of the update operation is the creation of an actual game resource. However, this new resource is a side-effect of updating an existing resource, not the outcome of a request specifically to create the game resource. Thus, you do not use the Location header field to communicate the resource URI as you do in normal create operations.

[6] Jim Webber, Savas Parastatidis, and Ian Robinson: "REST in Practice: Hypermedia and Systems Architecture", O'Reilly Media, 2010.

Instead, I use the `next:` field to return a hyperlink to the new resource, `/lobby/game/{game-id}`. This is an example of the *Hypermedia As The Engine of Application State (HATEOAS)*. I will discuss it below.

- As no new resource was created as result of the POST, the status code is 200 OK and the Location field is not used.

HATEOAS resemble human web browsing where web pages contain many hyperlinks and the user is free to click any to move to another page. In REST, hyperlinks in the same way allows the client to make state changes on the server side resources by doing CRUD operations on the resources that the server provides. To paraphrase our GameLobby, by Updating the future game, it makes a state change by creating the resource, `/lobby/game/{game-id}`, and provides the client with a resource identifier for it.

The *next:* key in my JSON above contains only a single resource, however, in more complex domains there may be multiple resources to operate on, so a list of resources may be provided instead. There is no agreed standard for how to name the section, but "next" or "links" are often used.

As both players have joined, it is visible that the game is ready, by reading the future game resource:

```
Read future game status
----------

GET /lobby/{future-game-id}

Response
  Status: 200 OK

  {
      playerOne: "Pedersen",
      playerTwo: "Findus",
      level: 0,
      available: true,
      next: "/lobby/game/{game-id}"
  }

  Status: 404 Not Found
  (none)
```

Thus, both players can now access a resource that represents the started game. At an abstract level, every game unfolds as each player makes some state changes to the game state, typically by moving pieces. However, whereas our OO game has a `move()` method, our REST game resource has not! It only has the CRUD methods, so to speak. This is the key challenge in making

REST level 2 designs: *how do we model complex state changes which are not easily represented by simple CRUD operations?* And, I might add, one of those that takes a while to get used to if you come from an OO background.

The key insight is to model the state change itself as a resource.

So, the above POST operation by Findus created a game resource on path "/lobby/game/{game-id}", however, as I also needs to make moves on this game, it should also create a *move resource* for that particular game: a resource defining one particular move to be made in the game. To rephrase this important insight, creating a game means creating *two* resources, one representing the state of the game, and another one representing the state of one or the next move.

I define the URI of move resources as "/lobby/game/{game-id}/move/{move-id}", and I define this resource's representation to be that of the relevant data to make an actual move. Let us take chess as an example, a chess game move can be represented by the following JSON (Chess notation uses the letters a-h for columns, and 1-8 for rows):

```
{
  player: "Pedersen",
  from:   "e2",
  to:     "e4",
}
```

That is, the above resource represents Pedersen moving the pawn on e2 to e4 on the chess board.

That is, white player can PUT the above JSON to the "/lobby/game/{game-id}/move/{move-id}" to make a state change on the move object and thereby change the game resource's state. So the procedure would be to

- PUT a valid game move representation on the move resource
- Verify that the PUT operation succeed (if the game move is invalid, the HTTP status code can tell so)
- GET the game state to see the updated game state

Thus, to **Story 3: Playing the game** both players can read the state of the game on the provided game resource

```
Read game status
----------

GET /lobby/game/{game-id}

Response
  Status: 200 OK

  {
      playerOne: "Pedersen",
      playerTwo: "Findus",
      level: 0,
      board: [ ... ],
      playerInTurn: "Pedersen",
      noOfMovesMade: 0,
      next: /lobby/game/{game-id}/move/{move-id}
  }

  Status: 404 Not Found
  (none)
```

This game resource represents the current state of the chess game: next player to make a move, number of moves made so far, the state of the board, etc.

Again, the noteworthy aspect is the `next` attribute whose value is the URI of the next move resource to update in order to make a state change to the game.

Let us continue with Chess as an example, so for Pedersen to make the opening pawn move, a PUT operation may look like

```
Make a Game Move
-----------

PUT /lobby/game/{game-id}/move/{move-id}

{
  player: "Pedersen",
  from:   "e2",
  to:     "e4",
}

Response
  Status: 200 OK

  {
    player: "Pedersen",
```

```
    from:   "e2",
    to:     "e4",
  }

  Status: 403 Forbidden

  Status: 404 Not Found
  (none)
```

Here, 403 Forbidden, can be used to signal that the move was not valid.

After a valid move, a new GET must be issued to fetch the updated state of the game - which player is next to move, how does the board look like now, and what is the `next` link which allows making the second move, etc.

Note that I have designed the resource URI `.../move/{move-id}` as a list of moves, typically the first move is made by PUT on resource `.../move/0`, the second on resource `.../move/1` and so forth. This way the move resource serves both as a HATEOAS way of changing the game's state as well as a historical account of all moves made, to be inspected by GET queries.

Discussion

The HATEOAS style departs fundamentally to a traditional OO style of design, and coming from an OO background, it takes quite a while to get used to.

The hard part is to stop designing in terms of methods that do complex state changes and to start design resources that represent the state changes instead. My presentation here has only scratched the surface of this architectural style, and if you are serious about REST, I highly recommend the "REST in Practice" book.[7]

Both *Broker* and REST supports remote communication between clients and a server. From a compositional design perspective, REST mixes a number of distinct responsiblities into a single role. A REST server handles the Request/Reply protocol over the internet (that is, the IPC layer of *Broker*), it dictates the encoding and object identity (that is, the marshalling layer of *Broker*), and it has no well-defined concept of the protocol with the servant objects (that is, the domain layer of *Broker*). Instead it relies on abstract design principles with few programming or library counterparts.

Thus, adopting REST is hard coupling. Compare to *Broker* where you may switch from a HTTP IPC to a MessageQueue based IPC, just be injecting new CRH and SRH implementations.

[7] Jim Webber, Savas Parastatidis, and Ian Robinson: "REST in Practice: Hypermedia and Systems Architecture", O'Reilly Media, 2010.

Another consequence of the role mixing is that TDD and testing REST architectures is less obvious: you do not have clearly defined IPC roles that can be replaced by test doubles as was the case with the *Broker*. What to do then, is the focus of the next section.

Another aspect that has some liabilities is the fact that HATEOAS dictate that returned resources have a notion of the state machine it is part of, which from a role perspective is an unfortunate mixing of domain knowledge and implementation dependent technicalities. An example is the Game resource, which models true game domain state, like involved players, state of the board, next player to take a turn, but *also* contains the `next` attribute which is a hypermedia link, a highly implementation dependent technical detail. There is not clear *separation of concern*.

Outline of REST based GameLobby

The presentation above is implemented in the *gamelobby-rest* project in the FRDS.Broker Library[8]. Note however, that the implementation there is only for demonstrating the HATEOAS principles, most domain code are just hard coded *fake-object* implementations.

I will not discuss the full implementation, but highlight the HATEOAS parts. The first aspect is the POST on the FutureGame resource, that creates both a game resource as well as the first move resource (error handling and other aspects have been omitted below):

```
1  post( "/lobby/:futureGameId", (request, response) -> {
2    String idAsString = request.params(":futureGameId");
3    Integer id = Integer.parseInt(idAsString);
4
5    FutureGameResource fgame = database.get(id);
6
7    // Demarshall body
8    String payload = request.body();
9    JsonNode asNode = new JsonNode(payload);
10
11   String playerTwo = asNode.getObject().getString("playerTwo");
12
13   // Update resource
14   fgame.setPlayerTwo(playerTwo);
15   fgame.setAvailable(true);
16
17   // Create game instance
18   int gameId = createGameResourceAndInsertIntoDatabase(fgame);
19
```

[8] https://bitbucket.org/henrikbaerbak/broker

```
20    fgame.setNext("/lobby/game/" + gameId);
21    updateFutureGameInDatabase(id, fgame);
22
23    return gson.toJson(fgame);
24  });
```

The code follows the standard template: demarshall incoming message, fetch relevant resource from storage, update state, and marshall and return resource. The important thing, however, is that the creation of the game resource *also* creates the (first) move resource. However, as the hypermedia link is embedded in the resource (in the class), this is hidden in the domain code:

```
1   private int createGameResourceAndInsertIntoDatabase(FutureGameResource fgame) {
2     int theGameId = generateIDForGame();
3
4     // Create the move resource storage
5     createMoveResourceListForGame(theGameId);
6
7     // Create the game resource
8     theOneGameOurServerHandles =
9             new GameResource(fgame.getPlayerOne(), fgame.getPlayerTwo(),
10                    fgame.getLevel(), theGameId);
11
12    return theGameId;
13  }
```

Though this code is only an initial sketch of a design, there are still two lessons learned here.

The first one is that though a given game resource and its associated list of moves are of course tightly coupled in the domain, they must be kept as disjoint datastructures in our REST scenario.

To see why, consider a classic OO design that would embedd the list of move resources within the game resource, as a field variable. Doing so, however, leads to the situation in which the JSON marshalling framework, like Gson, will embed the the full list into the returned resource, ala:

```
{"playerOne":"Pedersen","playerTwo":"Findus",
 "level":0,"id":77,"playerInTurn":"Pedersen","noOfMovesMade":2,
  "next":"/lobby/game/77/move/2",
  "moveResourceList":[
    {"player":"Pedersen","from":"e2","to":"e4"},
    {"player":"Findus","from":"e7","to":"e5"},
    {"player":"null","from":"null","to":"null"}],
  "board":"[...]"}
```

which is certainly not what I want. One can often tweak marshalling to avoid some fields in the resulting JSON but that is a slippery slope, and cost quite a lot of processing.

Second, as the two are kept separate from each other, the relation becomes weaker and the logic to keep them synchronized must reside outside the resources/objects themselves.

This also means that *making a move* then involves making changes to these disjoint data structures:

```
// Update the move resource, i.e. making a transition in the game's state
put( "/lobby/game/:gameId/move/:moveId", (request, response) -> {
  String gameIdAsString = request.params(":gameId");
  Integer gameId = Integer.parseInt(gameIdAsString);

  // Demarshall body
  String payload = request.body();
  MoveResource move = gson.fromJson(payload, MoveResource.class);

  // Update game resource with the new move
  GameResource game = getGameFromDatabase(gameId);
  makeTheMove(game, move);

  return gson.toJson(move);
});
```

The actual move is made in method `makeTheMove()` which necessarily have to fetch both the game resource and update it, as well as fetch the move list and update that.

In essense, the data model to implement in REST is more like a relational database with disjoint tables linked by keys, than the encapsulated object model known from OO.

7.11 Testability and TDD of REST designs

How can I implement the design above? Well, one way is to use manual testing and incremental development, similar to the PasteBin example. That is, implement a (Spark-Java) web server, that accepts a *One Step Test* operation. The obvious choice here would be POST on path /lobby as it is the starting point for the GameLobby system. Then use PostMan or Curl to verify that the web server accepts the request and returns proper JSON formatted resources.

The problem is that it is does not follow the TDD principle *automated test*, and regression testing is suffering - if I want to experiment or refactor, I have a bunch of Curl commands I have to execute to get to a certain state of the server. And none of it is within JUnit control.

Two options exists

- Use integration testing techniques: Spawn a REST server in the JUnit fixture, send HTTP requests as part of the test cases, and verify the return HTTP responses. Downside: Brittle tests that may be slow.
- Use the *Facade* pattern to encapsulate the REST paradigm using 3-1-2. Downside: There is more code to produce.

In the `gamelobby-rest` project in the FRDS.Broker library, the first option is chosen, as the simplicity of the gamelobby server does not warrent the larger setup required by using a *Facade*. But let us look at the second option.

The observation is that our web server acts as an intermediary between the requests arriving from the network, and our domain objects. For instance, a POST on path /lobby is a request, and the reply is an object that encapsulates statusCode, Location field, and a JSON body.

Thus we can make a facade that is as close as possible to the interface of the web server, for instance with a method:

```
1 public interface GameLobbyFacade {
2   public RESTReply postOnPathLobby(String restBody);
3   ...
4 }
```

and the initial TDD test may look like (in pseudo code):

```
@Test
public void shouldCreateFutureGameWhenPosting() {
  ...
  String body = "{ player: Pedersen }";
  RESTReply r = facade.postOnPathLobby(body);
  assertThat(r.getLocation(), is("/lobby/future-game-id-1"));
  assertThat(r.getStatusCode(), is(200));
  ...
}
```

The central requirement of the facade's methods are

- The methods should *closely* match those that the web server experience in all respects: parameters passed in, and return values. This is vital in order to make the code in the actual web server minimal and simple. The web server should *only* contain minimal code to extract parameters from the request, call the proper facade method, and return a response based on simple retrieval of values from the RESTReply object.

The central aspect is *simplicity in coding the web server* because the probability of getting it wrong is minimal. Still we need manual testing of this code (or some integration tests that actually spawn the server), but the less code to manual test the better, and if the code is simple, chances of errors are low.

The benefit of this approach is then:

- The facade allows TDD using the normal xUnit testing framework support.
- The facade allows automated testing of the core server code, i.e. the complex domain code that is vital to keep reliable during refactorings, and feature additions.

Of course, there are also liabilities involved

- The facade is a Java interface that does not match the REST/HTTP style perfectly, so I need to have helper objects, like the RESTReply interface/-class to represent a valid HTTP reply. These require a bit extra coding. Or you may use the classes from, say, the `javax.servlet.http` package, but still it requires some extra effort.
- If you TDD the facade before you have a good idea of the web server code, you may introduce a mismatch between what the web server actually have access to and what you have expected in the facade, requiring rework. So often it is a good idea so do some manual prototyping of the web server to minimize that risk.

7.12 Summary of Key Concepts

Representational State Transfer (REST) is an architectural style/pattern that model systems and information in terms of *resources* that can be named through *resource identifiers* using URIs. *Representations of data* are exchanged between *servers* and *clients* using standard formats, the *media types*. All interactions are *stateless*, so every request must contain all relevant information to process it.

At a more concrete level, resources are manipulated using the four CRUD verbs of HTTP: POST (create a resource), GET (read a resource), PUT (update a resource), and DELETE (delete a resource). For simple domains in which resources are independent and there is no need for transactions (updating several resources as one atomic operation) or complex state changes, this *Level 1 REST* is sufficient. The TeleMed case is an example, as it just deals with creating individual blood pressure measurements and reading them again.

For a more complex domain, you need *Level 2 REST* in which resources also contains references/hypermedia links to other resources that represents state changes. Thus, a returned resources to a client will have a link section with a set of resource identifiers that each can be manipulated using HTTP verbs. In our GameLobby example, a game resource also contains a "move" resource which can be UPDATEd to make a move, thereby indirectly affecting the state of the game.

My treatment of REST is highly inspired by Webber et al.'s book, *REST in Practice*[9].

Pedersen and Findus appear in the children books by Svend Nordquist.

7.13 Review Questions

Explain the three levels of REST usage in Richardson's model.

Outline the central concepts and techniques of Level 1 REST.

Outline the central concepts and techniques of Level 2 REST.

Explain how to model complex state changes using HATEOAS.

Discuss the benefits and liabilities of REST compared to *Broker*. Consider the programming model, and the *separation of concern* with respect to marshalling, IPC, and domain layer.

Discuss how TDD and automated testing can be made on a REST based architecture.

[9] Jim Webber, Savas Parastatidis, and Ian Robinson: "REST in Practice: Hypermedia and Systems Architecture", O'Reilly Media, 2010.