UNIVERSITY OF TARTU
Institute of Computer Science
Curriculum

Michael Ngugi Michuki

Exposing Rich Update Operations via REST APIs

Master's Thesis (30 EAP)

Supervisor(s): Luciano Garcia Bañuelos

Tartu 2017

Exposing Rich Update Operations via REST APIs

Abstract:

In the past few years Representational State Transfer (REST) has emerged as a leader of modern Application Programming Interface (API) design for its simplicity and versatility. This has facilitated APIs with URLs that make use of HTTP methods like GET, POST, PUT and DELETE. It has also led to producing intuitive models for client developers. However, there are two issues that REST doesn't solve alone, the first issue being standardized responses. Most enterprises have their own custom APIs, usually a JSON response that maps clearly to their custom data model. A good example is when a Twitter API client is not able directly communicate with a Reddit API and vice versa. This leads to numerous API client applications that do almost, but not quite the same thing. Hence we find numerous developers duplicating the same efforts. The second issue is API linking. As put by the World Wide Web Consortium (W3C): JSON doesn't include built-in support for hyperlinks, which fundamentally constitutes as a building block on the Web. Hence the drawbacks of the two is that the API endpoints are only linked by API documentation, thus users are forced to peruse through pages of API documentation to comprehend the relationships between the API endpoints so as to understand the actions that are required to interact with a given resource. In summary, we find that in modern day applications, the APIs developed have their update operations mapped to methods that directly manipulate the domain. In this research we discuss the underlying problem with update queries. We also unearth better ways to enhance APIs to expose richer update methods using Domain Driven Development (DDD) and Command Query Responsibility Segregation (CQRS). The first contribution in this research is to take a hypermedia format namely Collection +JSON, and to enhance it to accommodate and expose multiple update commands and templates through a single REST update method dynamically, both in the API and the client application. In simple words, we reconcile the simplicity of REST, with the ample concepts of DDD and CQRS. We also implement an annotation based resource assembler that generates commands dynamically by using a generic controller and interface. This reduces the implementation needed to dynamically inject domain service methods into the API.

Keywords:

JSON, API, REST, DDD, URL, HATEOAS, CQRS, CRUD

CERCS: P170

Rikkalike uuenduste avaldamine käskluste kaudu REST APIs

Lühikokkuvõte:

Paari viimase aastaga on Representational State Transfer (REST) muutunud Application Programming Interface'i (API) disaini liidriks tänu lihtsusele ja mitmekülgsusele. See on hõlbustanud API'sid URL'ide juures, mis kasutavad selliseid HTTP meetodeid nagu GET, POST, PUT ja DELETE. See on viinud lihtsustatud mudelite toomiseni klientide tarkvara arendajatele. Siiski on kaks probleemi, mida REST ainuüksi ei lahenda. Esimene neist on standardiseeritud vastused. Enamikel ettevõtetest on oma API'd, tavaliselt JSON tüüpi vastusega, mis ühtib nende andmemudeliga. Hea näide on see, kui Twitter'i API klient ei ole võimeline otseselt ühendust võtma Reddit API'ga ja vastupidi. Seetõttu teevad paljud API kliendid peaaegu sama asja. Sellest tulenevalt leiame me olukorra, kus arvukad arendajad kordavad sedasama saavutust. Järgmine probleem on API'de sidumine. World Wide Web'i Consortsium'i (W3C) kohaselt JSON ei sisalda sisseehitatud toetust hüperlinkidele, mis moodustavad põhilise Web'i ehitusbloki. Nende kahe puudused tähendavad seda, et API lõpp-punktid on seotud API dokumentatsiooniga, seega on kasutajad sunnitud hoolikalt läbi lugema mitmeid lehekülgi API dokumentatsioone, et mõista API lõpp-punktide vahelist suhet ning saada aru, mis tegevused on vajalikud juurdepääsuks antud vahendile. Kokkuvõttes me näeme, et kaasaegsetel rakendustel, arendatud API'd, on uuenduste käsklused ühendatud meetoditega, mis manipuleerivad otseselt domeeniga. Selles uuringus arutleme uuendusi otsivate käskluste probleemi üle ja püüame leida võimalusi, et täiustada seda rikkalike uuendusmeetodite ja erinevaid lähenemisi kasutavate, uuendusi otsivate, käskluste kaudu. Esimese panusena, selles uurimustöös, tuleb võtta hüpermeedia formaat Collection +JSON ja parendada seda nii, et see mahutaks rohkem uuenduste käsklusi, läbi üheainsa REST uuenduse meetodi, dünaamiliselt, nii API's kui kliendi tarkvaras. Lihtsalt öeldes me kooskõlastame REST'i lihtsuse Domain Driven Development and Command Query Responsibility Segregation'i (CQRS) rikkalike kontseptidega. Me loome märkustel põhineva ressursi pakkija, mis genereerib käsud dünaamiliselt kasutades jagatud kontrollijat ja liidest, mis vähendab koodi, mida vajatakse laadimisel valdkonna teenuse meetodina API'sse.

Võtmesõnad:

JSON, API, REST, DDD, URL, HATEOAS, CQRS, CRUD

CERCS: P170

TABLE OF CONTENTS

Table of Figures

1 INTRODUCTION

Representational State Transfer (REST) as coined by Roy Fielding [1] became popular over the years for offering an easier way to integrate foreign applications to share resources using a set of rules referred to as an Application programming interface (API). APIs use a set of Create, Read, Update and Delete (CRUD), to provide an abstraction of data manipulation in data driven applications such as database systems and applications in the World Wide Web (WWW). APIs do this by sending or requesting for a resource representation, which is called a JavaScript Object Notation (JSON) payload. The payload is usually the result of the execution of the CRUD operations by REST architectures.

The WWW which is made of a large number of sites, is a collection of interconnected documents and resources, linked by hyperlinks and Uniform Resource Locators (URL) accessed through the internet, which is a global data system of hardware and software infrastructure that provides connectivity between computers. These sites run on numerous different server implementations and experience intermittent upgrades. As mentioned by Bajali [2], similar to the way a person visits a website, a REST client application accesses the root API URL and after that uses the server-provided links to dynamically find accessible actions to the resources it needs. The client does not need to have known of the service or the different steps involved in a process flow. In fact, the client no longer has to have any hard coded URL structures in order to gain access to resources. This facilitates frequent URL changes by the server for different resources which allow the evolution of the API without breaking the clients. It is through Hypermedia as the Engine of Application State (HATEOAS) that this interaction is possible. HATEOAS should be used to find your way through the API.

1.1 Context

Unfortunately most of the APIs today do not make use of all the solutions provided by HATEOAS. We find that large applications with millions of lines of code, like banking applications and social networks, require a lot of upkeep and have numerous integrations, which make developers spend hours perusing through large documentation in order to create new features or make use of particular features in the API. In fact, as mentioned by Mike Amundsen [3], most APIs today, once deployed, cannot change and end up being static for years, depending on the complexity of the system which should be adaptable to change. Amundsen [3] proceeds to state the following:

- An API that is important will remain accessible for a number of years because of its numerous clients. Regardless of the possibility of the domain changing once in a while, the combined impact on customers can be enormous.
- Some APIs change constantly, with new components and business rules. While in some APIs, every customer can customise the processes to suit their individual needs. Regardless of the possibility that the API itself never shows signs of change every customer will experience it in a different way.
- The team that writes the API clients, in most cases is not on the same one that writes the servers, hence changes to the API must be handled with care because of the possibilities of breaking unknown clients.

We find that in today's industry many developers depend particularly on popular or experimental techniques and facets to build their own software APIs. However, with no standardized approach many different techniques are employed based on the problem at the point of development and the solution to be developed. [4] Some developers may use techniques like Test-driven development (TDD), which implies writing a test that fails before writing a working program. Other developers employ Domain Driven Development (DDD), which uses

the domain data and business rules to shape the application, among other techniques which may also be used hand in hand.

DDD over the years became popular in many applications because its bridges the gap between developers and domain experts by promoting the use of the common vocabulary (cf. ubiquitous language) taken from the application domain itself such as accepting an order, rejecting an order etc. In simple terms the domain is the inner workings of an organization or a process that yields specific results e.g. In a Bank, any transaction like the withdrawal of money from a customer's account. DDD also relies on the use of object oriented design in which methods, classes and objects reflect concepts in the domain. Particularly, DDD promotes the use of "Intention-Revealing Interfaces" as a practice in the design and coding, which is translated for instance in the use of self-explanatory method names e.g. deductTaxesAfterSalaryIsPaid which is used to reduce the taxes in a salary. Unfortunately, most developers do not follow this approach and they implement methods that mimic low level operations like getters and setters e.g. getOrder as used in assembly languages.

Additionally, the operations that modify the state of the resources are so diverse that exposing these operations by means of a REST interface, or by the use of a simple operation becomes complex. Thus a better approach, as suggested in this research, would be to reconcile the simplicity of REST with the rich concepts of DDD. In order to bridge this gap, it is important to understand the strengths and weaknesses of the existing JSON Hypermedia formats i.e. HAL, SIREN and Collection JSON. Hypermedia as explained by Amundsen [3] is an extension of hypertext that providing access to multimedia facilities, like documents, sounds and video. This information is useful in the selection of the most suitable representation that generates a JSON payload that supports linking (hyperlinks) and also makes it easier to navigate without having to understand the entire meaning of the API. My contribution is to take one of these representations (Collection +JSON) and enrich it in a way that allows accessing the diverse update operations using the HTTP protocol seamless and to make the server side and client side more robust.

Unfortunately, one of the biggest pieces missing from common Hypermedia types is the ability to dictate what requests can be made to alter the application state. According to Sookocheff [5] we are able to offer links between APIs, potential actions, endpoints and documentation by making use of Hypermedia in JSON responses. This allows for APIs that are readable and clear from the response by providing the set of available actions that a client may want to take to change the state of a resource. Additionally, one is able to adopt the standard without introducing breaking changes to the API. [5] This augmentation mostly serves as a way to self-document the API. It facilitates decoupling the responsibility of always either handling the domain operations in the backend or frontend by exposing the actions one can take on a given resource as hyperlinks. The operations in the REST controller can then be further packaged into a generic controller that contains a single update operation that facilitates executing the operations in service by the name. Without this flexibility an additional effort to update the corresponding front end applications becomes adamant when a new version of the API is released. The final result of resolving these ideas is a richer and readable API.

1.2 Proposed Approach

In this research the main objective is to reconcile the simplicity of REST APIs, with the rich concepts of DDD and CQRS. Roy Fielding [6] , built and formulated REST by studying the principles that made the Web work. This set of principles were documented in his dissertation that, when applied to a system, made it as resilient as the Web. In Chapter 5 of his dissertation, Fielding outlined the four REST interface constraints. Initially, the identification of resources, which means that anything with a label such as a video, a document or a file is a resource and can normally be accessed using a URL. Secondly resources can be manipulated using representations because the server can consume and respond with a representation of the resource with a modified state. Finally, the media that a client chooses to interact with should be self-descriptive enough to provide related resources and actions that can be accessed for other results. All the three constraints put together are what form Hypermedia as the engine of application state.

In order to achieve a robust API, it is important to select a JSON representation format that promotes the constraints of REST. Unfortunately, one of most common approaches used today is the use of (Remote procedure call protocol) RPC [1] -style resources such as /api/conference/{id}/rename. As mentioned by Ali [7], this approach goes against REST's resource-oriented presentation even though it seems to remove the need for the arbitrary verbs. It is important to note that a resource is a noun, the HTTP verb is the action or verb and self-descriptive messages are used to convey other angles of information and intent.

In order to demonstrate the transformation, the first challenge will be to adapt the applications domain layer by creating meaningful methods. This is achieved by creating methods that package all low level methods into a self-descriptive method that clearly communicate its intention e.g. confirmClientPayment. It is important to follow the principles of DDD in order to have a domain layer that is easily understandable. As promoted by CQRS, we then implement a generic abstract controller class that already implements the redundant CRUD HTTP methods. This is important because it can be inherited by any controller class and it can be reused as opposed to rewriting the same HTTP methods in every single controller class. We then remove and replace the redundant HTTP method calls with a single instantiation of the abstract controller. Among the parameters passed during instantiation is the domain service. The domain service is then injected automatically through the interface and is used by the controller to access its methods. Following the principles of CQRS, we will first reorganize the different methods in to commands and queries. We will then map the commands to either PUT, PATCH or DELETE and the queries to GET. In order for this to work throughout all the controllers, the main update HTTP methods will facilitate the access of domain methods by name which is passed as a header. Commands in this situation will be used to change the state of a resource e.g. cancelSeatReservation which is used to cancel seat reservations, while queries will be used to filter or search for specific information e.g. findAvailableSeats.

In summary, the scope for this research will focus on the update operations. It is adequate enough to have a command in the payload of the HTTP message so as to express any arbitrary action. The REST API is not just another layer on top of the persistence layer. It provides access to the richness of the business domain, its operations and workflows. Hence, it is possible and preferable to show the intent without resorting to custom verbs.

[1] https://en.wikipedia.org/wiki/XML-RPC

1.3 Objectives

The main research question that this research tries to answer is how to expose richer update operations in REST APIs and develop applications that make use of REST's simplicity of having true CRUD operations that are handled individually under one HTTP request.
Below is a summary of the scope and objectives of this research:

- Refactoring a plain JSON API into one that implements Hypermedia and implements the constraints of REST e.g. by exposing self-descriptive domain methods in the API as hyperlinks.
- Packaging all basic CRUD operations in to a single reusable generic and abstract controller.
- Implementing a single PATCH or PUT HTTP Method that facilitates access to all update operations in the domain layer.

1.4 Scope

Below is a summary of the scope of the thesis:

- **Background and Related work**– provides a background in REST services, Hypermedia and the state of the art. It also looks into related concepts and how existing solutions have solved the same problem.
- **Solution and Contributions** – this section describes the contributions and the steps taken to arrive to a richer API.
- **Evaluation** – this section evaluates and validates the implemented solution it also discusses the advantages and disadvantages of the proposed solution.
- **Conclusion** – this section provides a summary of the thesis as a whole and how it can be used to solve real life problems. It also provides some suggestions for future work.
- **References** – provides the appendix and the resources used to generate this report.

2 BACKGROUND

In this section we will discuss the background of our research to unfolding the various concepts to be used in our project. Firstly, we unfold the problem by scoping out the context, followed by Domain Driven Development, CQRS, REST, HTTP methods and finally Hypermedia and its respective formats.

2.1 Context: Creating an Order

Contoso[2], a Conference management system adopted from Bets [8], is used by customers to host and manage conferences at specific physical locations. The customers have the ability to create a conference, define its characteristics and manage sales of the different seat types at the conference. A customer (the registrant) begins the process by creating an account in the system before they are able to create and manage their conferences or purchase seats at a particular conference. Once registered, a registrant is able to create a new conference and manage the information of the conference such as the conference name, description, start date and end date. Additionally, he can define the type of seats (e.g. VIP, regular), and, for each type, the number of available seats in a given conference. Later, the registrant can also make a conference

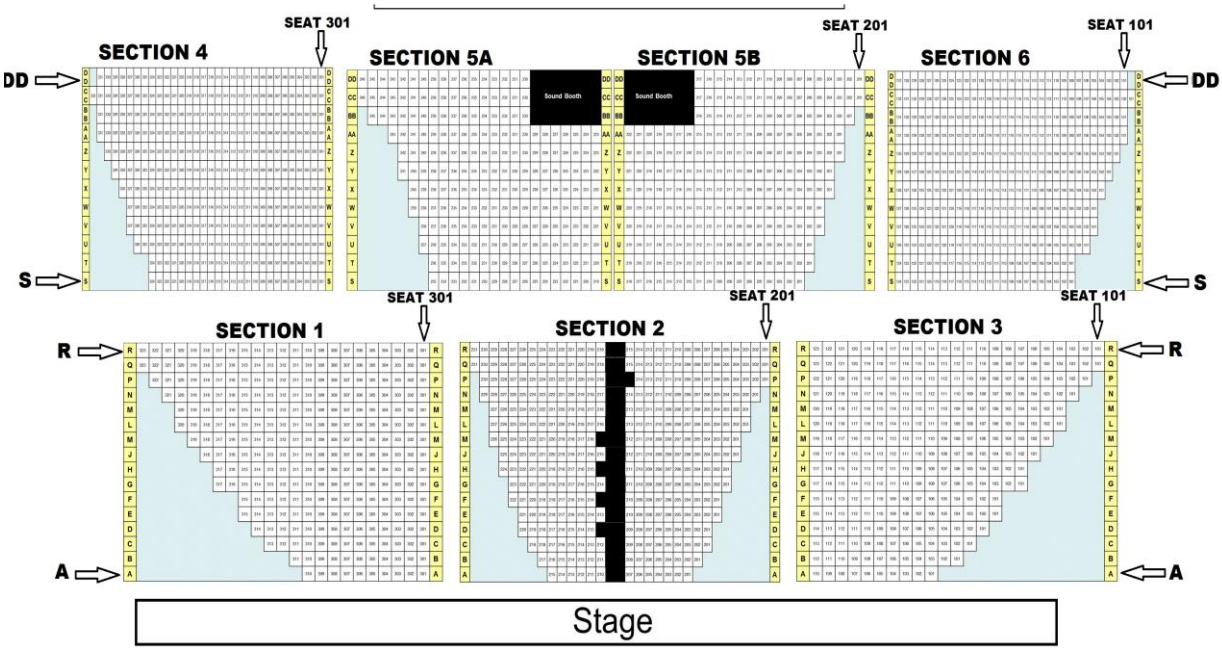


Figure 1 The Conference map

visible/invisible by publishing or un-publishing it. The registrant is also able to specify events at a conference such as training workshops, meetings and ceremonies for which an attendee must have a ticket.

The purchased seats are assigned names of attendees by the registrant. The system creates a purchase order to manage the reservations and payments made when the registrant interacts with the system. The process of Ordering is in two stages: first, the registrant begins by reserving a number of seats and then continues to pay for the seats and confirm the reservation. If the registrant does not complete the payment, the seat reservations expire after a fixed period and the system makes the seats available for other registrants to reserve. An order can contain multiple order items. An order item in this scenario represents a seat type and quantity. An order item is first set to pending state if the system has reserved the quantity of seats of the seat type

[2] The current scenario is inspired by Microsoft CQRS journey guide

requested by the registrant. An order item is in the cancelled state if the system cannot reserve the quantity of seats of the seat type requested by the registrant as seen in Figure 1.

A seat grants an attendee the right to access a conference for a specific session at the conference such as a celebration party or a workshop. In the conference rooms a seat is mapped and located by a unique identifier which consists of a section, a row letter and a section number. Rows are marked with letters of the alphabet i.e. A to Z. The system automatically tracks the seat availability for each type of seat. Initially, all seats are available for reservation and purchase. The system allows the user or the registrant to select a seat based on the seat type. When a seat is successfully reserved, the system automatically decrements the number of available seats. If the system cancels the reservation, the number of available seats of that type is incremented. The system automatically manages seat sale to make sure that the bookings are not oversubscribed. This system also operates a wait-list that reallocates the seats if other attendees cancel. The orders in the waiting lists are updated to pending state.

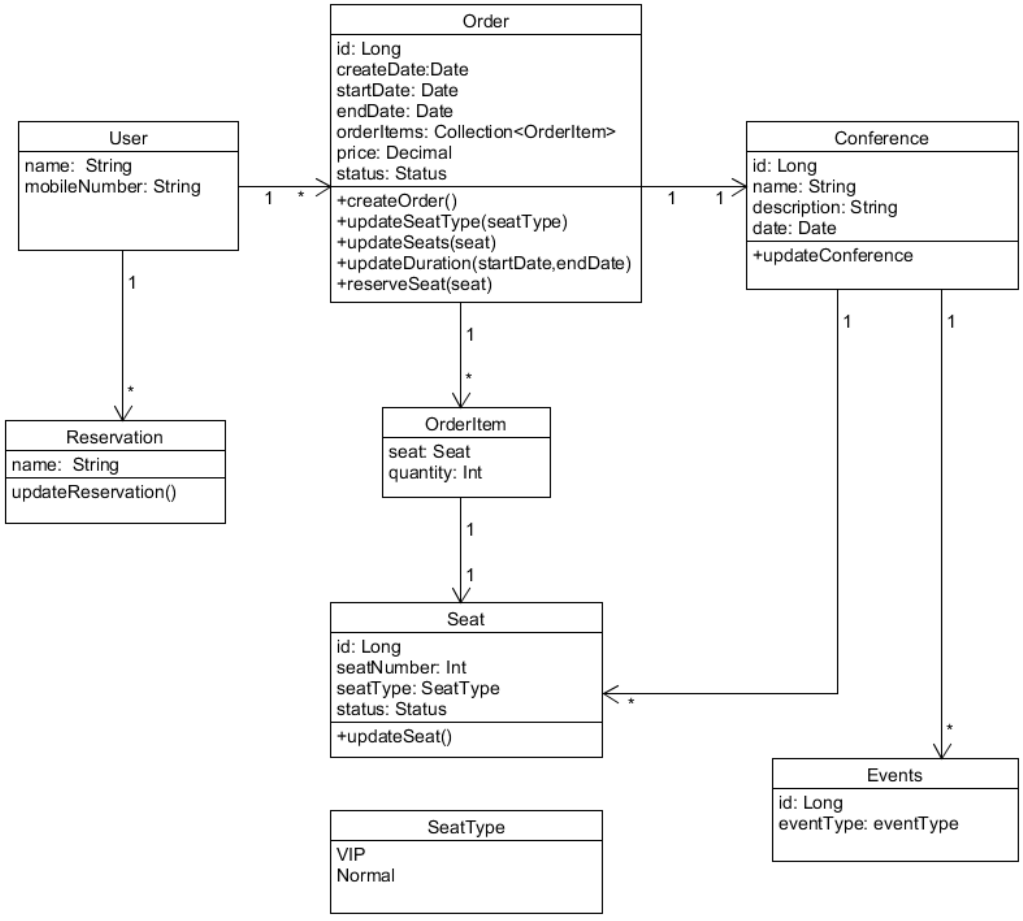


Figure 2 Contoso Ordering System Domain Model Diagram

A reservation on the other hand, is temporary for one or more seats at a conference. Below is the lifecycle of a reservation which when a registrant begins, the reservations are set to pending state. When the seats are confirmed, the system updates them to approved state and makes them unavailable for other registrants to reserve. The reservations are held for 15 minutes, till the registrant is able to complete the ordering process by paying for the seats. If the registrant fails to pay for the tickets within the 15 minutes timeframe, the system updates the reservation to closed state and the seats are made available for other registrants to reserve. A registrant is able to assign different users such as a speaker or an attendee with a seat on a confirmed order of a

conference. On the other hand, if the payment is confirmed, the system automatically updates the reservation to booked state and it automatically processes the invoice to be sent to the attendee. Once it is able to send the invoice successfully, it updates the reservation to invoiced state.

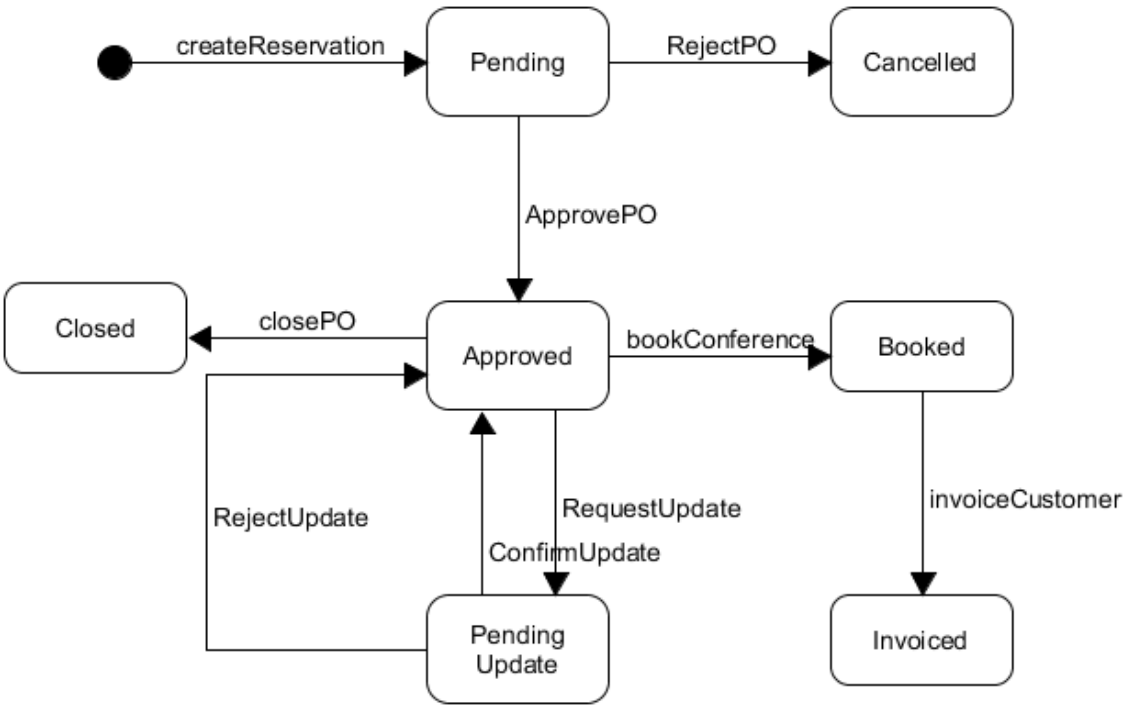


Figure 3 Resource lifecycle of a Purchase Order

Our context however will focus solely on the ordering process by selecting a few update operations to demonstrate the various complexities that come with updating a seat or an order with more or less seats. We will then present the different Hypermedia formats i.e. such as HAL, Siren and Collection +JSON. In typical Hal based applications the domain model has its business update operations mapped out as individual update queries as seen below. They are as follows

- updateSeatType(SeatType seatType)
- requestPurchaseOrderUpdate (Date startDate, Date endDate)

2.2 Representation State Transfer (REST)

Roy Thomas Fielding (2000) came up with the term Representational State Transfer shortened to REST in his research [1] as an architectural style with a set of architectural constraints such as being stateless, layered systems, cacheable with a client/server relationship, and a uniform interface aimed at designing internet-scale distributed systems. It is elaborated as follows:

1. **Uniform Interfaces** They define the interface between clients and servers. They simplify and decouple the architecture which enables independent evolution. Below are the guiding principles :
   - **Resource-Based Individual resources** are found using URLs in requests as resource identifiers and conceptually separate from the representations that are returned to the client.
   - **Self-descriptive Messages** - Each message contains a description of how to process it such as the context or type.
   - **Hypermedia as the Engine of Application State (HATEOAS)** it is defined as the way clients and servers deliver state to each other either via the body content, response codes, response headers, query-string parameters, request headers and the requested URL (the resource name). This is also called Hypermedia or hyperlinks within hypertext. It also means that links are located in the returned body or headers to expose the URL for fetching of the object.
2. **Statelessness** This is essential and necessary in order to handle the requests contained in the request itself, whether as part of the body, query-string parameters, URL, or headers. The URL uniquely identifies the resource while the body contains the state of that resource. After processing, the appropriate state is sent back to the client via headers, status and response body.
3. **Cache-ability** clients are able to cache responses either implicitly or explicitly, to avoid reusing state in response to further requests. Some proper caching partially or completely eliminates some client–server interactions, further improving scalability and performance.
4. **Client–server separation** the uniform interface separates a client from a server which means that clients do not deal with data storage, and that it remains internal to each server so that the portability of client code is improved. The simplicity allows it for them to be respectively maintained independently.
5. **Layered system** a client is not aware of whether it is connected directly to the end server or to an intermediary along the way. Intermediary servers may improve scalability using load-balancing and shared caches.

2.3 CRUD Operations

In the context of this research we will look at the various operations that are used. The widely-used HTTP verbs are POST, GET, PUT, and DELETE and they correspond to create, read, update, and delete (or CRUD) operations, respectively. There are other verbs but they are not used that much. Out of those less-frequent methods, OPTIONS and HEAD are used more often than others.

- **GET**

This HTTP method is used to retrieve information or a representation of a resource. GET requests must be safe and idempotent, meaning that the results are the same despite multiple fetches with the same parameters. In case of errors, a 404 (NOT FOUND) or 400 (BAD REQUEST) is often returned. It is not advised to expose unsafe operations via GET like modifying a resources on the server. [7]

Below is an example request response of a Collection +JSON. When a user executes a HTTP GET request to the URL /orders of a collection, a 200 OK response together with a Collection +JSON document is returned with one or more objects in an array. [3] The response may describe all, or only a partial list, of the items in a collection.

```
- REQUEST -
GET /orders HTTP/1.1
Host: www.contoso.com
Accept: application/vnd.collection+json

- RESPONSE -
200 OK HTTP/1.1
Content-Type: application/vnd.collection+json
Content-Length: xxx
{
        "Collection”: {...}
}
```

- **POST**

According to Amundsen [4], The POST verb is mostly used for creating new or subordinate resources such as a parent resource. In simple terms, when creating a new resource, POST to the parent and the service should associate the new resource with the parent, assigning an ID i.e. new resource URL etc. Below is an example request response for Collection +JSON. When a user client executes a HTTP POST request with data based on the template provided in the API to the URL of a collection, a Collection +JSON response is then returned with one or more objects in an array. Below is an example of a response. The response it may describe all, or only a partial list, of the items in a collection. [4]

```
- REQUEST -
POST /orders/ HTTP/1.1
Host: www.contoso.com
Content-Type: application/vnd.collection+JSON

{ "template" : { "data" : [ ...] } }

- RESPONSE -
201 Created HTTP/1.1
Location: http://www.contoso.com/orders/12345
```

Once the resource is created a location header with a link to the newly created resource with the 201 HTTP status is returned. POST is not usually safe or idempotent hence it is recommended for non-idempotent resource requests. It can also be used to update an entity.

- **DELETE**

The Delete method is used in most scenarios to remove an entity. The entity may be deleted permanently or it may be visually eliminated from a record stack, but still remain as a record in the database. For example as seen with Collection +JSON. When the client requests for a HTTP DELETE request to a URL of a collection, the response that is returned describes the deleted entity. If the delete request is successful, the server usually responds with a 204 HTTP status code. [4]

```
- REQUEST -
DELETE /orders/12345 HTTP/1.1
Host: www.contoso.com

- RESPONSE -
204 No Content HTTP/1.1
```

- **PUT (UPDATE)**

PUT replaces an existing entity if a subset of data elements are provided while the rest is replaced with empty or left unchanged. PUT is an idempotent request that is usually used for updating, "PUT"-ing to a known resource URL with the request body having the newly-updated representation of the original resource. However, PUT can also be used to create a resource if the identifier is created on the client side. Alternatively, POST is what is used to create new resources and to provide the client-defined ID in the body. [7]

```
- REQUEST -
PUT /orders/12345 HTTP/1.1
Host: www.contoso.com
Content-Type: application/vnd.collection+JSON

{
        "template":{
                "data":
                        [{"name":"startDate","value":"2016-03-12"},
                        {"name":"endDate","value":"2016-03-22"}]}
}

- RESPONSE -
200 OK HTTP/1.1
```

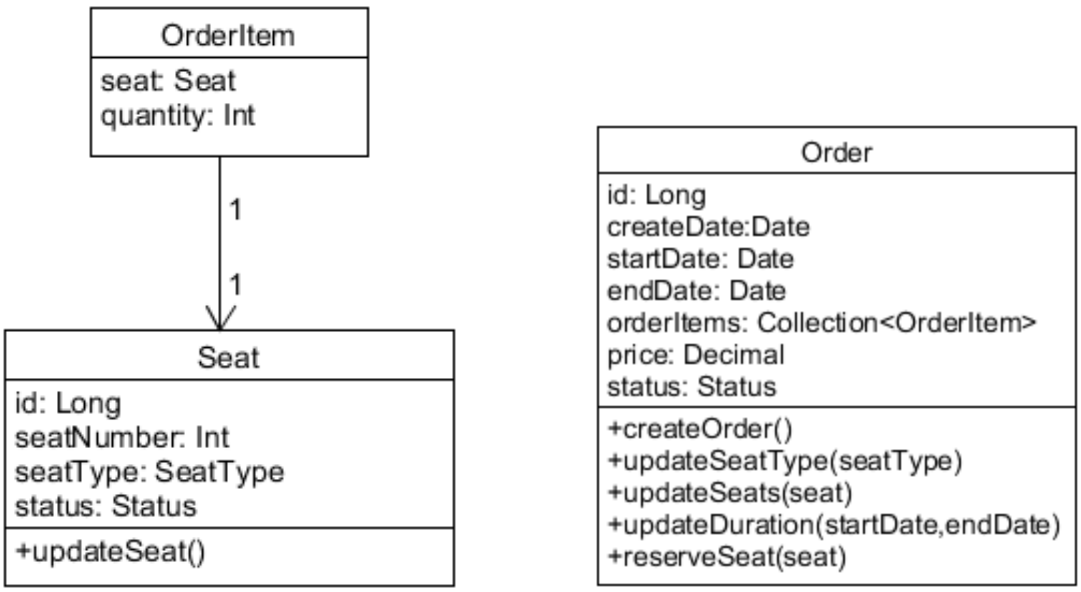
Once the request is successfully updated, the server responds with a 200. PUT is not a safe operation, in that it modifies the state on the server, but it is idempotent. If you update a resource using PUT and then call the same request again, the resource will still be there in the same state as when you called it first. However if for example calling PUT on a resource increments a counter within the resource, then the call is no longer idempotent. It is recommended to keep PUT requests idempotent and use POST instead for non-idempotent requests.

2.4 HATEOAS

HATEOAS is an acronym for (Hypermedia as the Engine of Application State) [9]. It means that the media type used should be a Hypermedia format. It should at least have the notion of a link and that the client requires no prior knowledge beyond a generic understanding of the Hypermedia format to interact with the server. As described by Mauro [10], just as a web browser is only required to know HTML and the URL of the homepage of a website, an API client only needs to know a supported media type and the URL of an entry point to the API. It can then follow hyperlinks to dynamically scan the API's resources. [10] With each HTTP request, the state of the application, as expressed by the client's current URL. No further out-of-band information, such as schemas or API descriptions, is needed. A Hypermedia format has to define a way to construct valid state transitions from a server response. For example, the HTML format specifies how to construct a GET request from a link element and its Href attribute, as well as how to construct a POST request from a form and its containing elements. HATEOAS enables very loose coupling between the client and the server. [3] Also, similar to a web browsers ability to navigate a website using HTML, generic clients interact with APIs that communicate in a Hypermedia format that is supported by the client.

2.5 Hypermedia Formats

To elaborate on the differences in the Hypermedia formats, we use the orders API for managing an Orders Resource derived from the Order class used by the Conference Management system. The Order resource can be expressed with this simple diagram.



Below is a HAL representation of what would be yielded as a typical JSON for the order when one fetches a list of orders.

```
GET http://contoso.com/orders
{
    "_links": {
        "self": { "href": "http://contoso.com/orders/1234" },
        "next": { "href": "http://contoso.com/orders/1234?page=2" }
         "cURLes": [{ "name": "doc","href":
         "https://contoso.com/docs/rels/{rel}", "templated": true }],
        "doc:purchaseorders": { "href": "/how-to-create-new-orders/" }

    },
    "size": "2",
    "_embedded": {
        "order": [
            {
                "_links": {
                    "self": { "href": "http://contoso.com/orders/4" },
                    "purchaseorders": {
                    "href": "https://contoso.com/orders/1230" }
                },
                 "idres": "4"
                 "name": "Alex wong",
                 "price": "490"
            },
            {
                "_links": {
                    "self": { "href": "http://contoso.com/orders/5" },
                    "purchaseorders": { "href":
                    "https://contoso.com/orders/1234" }
                },
                 "idres": "5",
                 "name": "Lori Stan",
                 "price": "590"
            }
        ]
    }
}
```

Figure 4 HAL JSON response of a list of Orders.

2.5.1 HAL: Hypertext Application Language

HAL is a simple lightweight and consistent format that uses the idea of Resources and Links to model its JSON responses. As described by Amundsen [11], HAL is simple to use and easy to understand making it one of the most popular Hypermedia types in APIs today because one can customize it to their own needs.

The HAL Model

In a typical HAL model, the Resources usually have their state defined by key-value pairs, Links (which lead to additional resources) and Embedded Resources, which are children of the current resource, are embedded in the representation as explained by Sokochef [5].

Resources

Resources usually have a self-link URL which is represented via a 'self' link in most cases as seen below:

```
 {
    "_links": {
        "self": { "href": "/rest/orders" }
    }
}
```

Resources comprise of:

- Links (to URLs): This is a combination of URL (Uniform Resource Identifiers) that provide identify a resource and also provide a hyperlink access to other resources.
- Embedded Resources (i.e. collection of resources): They are resources that are fully contained in the resource. E.g. every Order is embedded in the JSON representation as an Embedded Resource. These
  Resources supplement the current resource state with additional, related resources such as a list of items.
- State (standard JSON or XML data): State is the traditional JSON key-value pairs defining the current state of the resource.

Links and Link Relations

A JSON HAL object usually has a collection of links named _links which contains the name of the link and it describes the relationship between the resource and the link. The _links property also usually contains a self-entry pointing to the current resource. [11]

Kelly [11] describes the Links, as seen in figure 5, as an object that comprises of a relation i.e. 'rel' which is used to indicate the purpose or meaning of a particular link. Link rels, which are usually a key within the _links hash map, are normally used to distinguish between a resource's links and to associate the link meaning i.e. the 'rel' with the link object that contains additional data like the actual 'href' value. It is also possible to add a link for the Order’s resource under the _links property so as to retrieve the full list.

Links comprise of:

- A target i.e. a URL
- A relation also known as the 'rel' is the name of the link
- Optional properties which are used to facilitate with versioning, deprecation, content negotiation, etc.

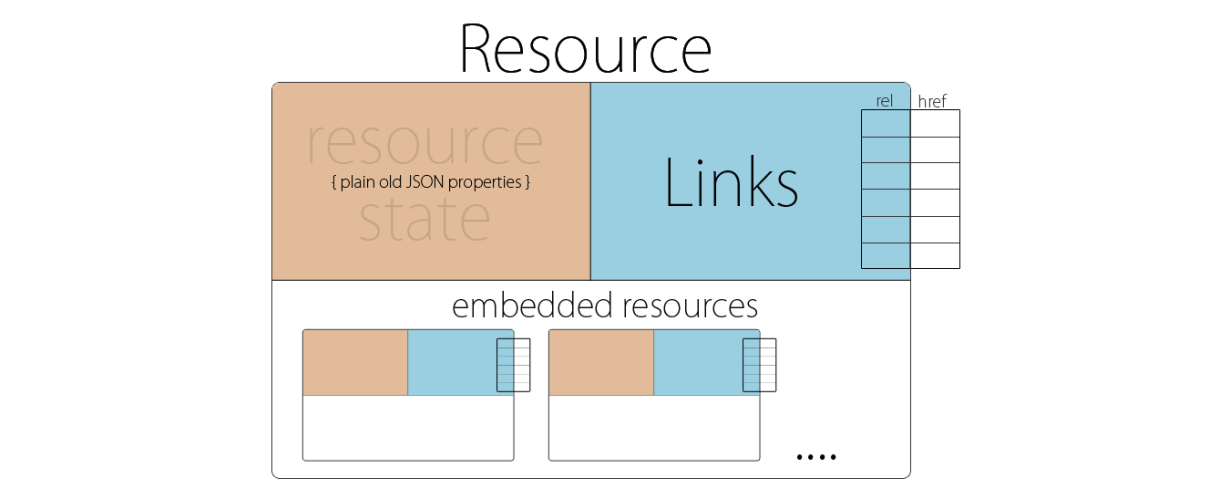


Figure 5 Structure of a HAL Resource [3]

API Discoverability

Link rels should be URLs used to provide documentation about a particular given link, which makes them "discoverable". Since URLs are rather too long to use as keys HAL makes use of tokens called "CURIEs", which are defined in the document and are used to express link relation URLs in a simpler and more compact fashion. Curies are usually expanded by post fixing the curie name with a colon: followed by the name of the resource as seen in Figure 7. i.e. doc:purchaseorders instead of using a long URL like http://contoso.com/docs/how-to-create-new-orders.

Most simple APIs today follow a simple uniform mapping structure where URLs are mapped directly to method names that are defined in the controller as shown below. Every HTTP method is mapped to a Verb that coincides with the intended action. i.e. GET is used to map methods that fetch data, while POST is used to send data to the server for processing.

HTTP Method	URL	Payload	Result
POST	/api/orders	CreateNewOrder	Creates New Order Item
GET	/api/orders/{id}	GetOrder	Returns Order Item
GET	/api/orders	GetOrders	Get All Orders
PUT	/api/orders/	UpdateOrder	UpdateOrders
PUT	/api/orders/updateSeat	UpdateSeat	Update order seat information.
PUT	/api/orders/requestPOUpdate	UpdateDuration	Change start or end dates.
DELETE	/api/orders/{id}	DeleteOrder	DeleteOrders

Table 1 HTTP methods in a HAL based system

[3] http://stateless.co/hal_specification.html

Persistence and integration

The process of Integration in a simple application is fairly easy. In order for a developer or a client to integrate and make use of the operations made in an API, one has to put read and write semantics in the client applications code. This has to be aligned with the API documentation that is normally provided by the API provider. Below are the basic CRUD operations and how they respond in a HAL application.

- *Adding an Item using POST*

In order to create a new item one must submit the required parameters in JSON format. Once the client's submission is successful, the system receives the JSON data, processes and replies with a status 201 response together with a location header that has a URL of the newly created item resource. Once an item resource has been created and its URL is known, the URL [4] can be used to read, update, and delete the resource.

```
- REQUEST -
POST /customer/ HTTP/1.1
Host: www.contoso.com
Content-Type: application/hal+JSON

{
    "firstName": "Michael",
    "lastName": "Michuki",
    "age": "26"
}

- RESPONSE -
201 Created HTTP/1.1
Location: http://www.contoso.com/customer/1
```

- *Reading an Order using GET*

Clients usually retrieve an existing item resource by sending an HTTP GET request to the URL of an item resource. If the request is valid, the server will respond with a representation of that item resource. The figure below is an example of how the response may describe all or partially, the items in a collection[5].

A Typical response includes a list of items in the collection, however for a single resource the properties of each element are given by explicit name/value pairs within a data attribute as shown below.

```
- REQUEST -
GET /orders HTTP/1.1
Host: www.contoso.com
Accept: application/hal+JSON

- RESPONSE -
HTTP/1.1 200 OK
Content-Type: application/hal+JSON

{
  "_links": {
  "self": { "href": "/orders" },
  "next": { "href": "/orders?page=2" },
  "find": { "href": "/orders{?id}", "templated": true }
```

[4] http://amundsen.com/media-types/collection/format/#types-URL
[5] http://amundsen.com/media-types/collection/format/#objects-collection

```
      },
      "_embedded": {     "orders": [{
         "_links": {
         "self": { "href": "/orders/1234" },
         "customer": { "href": "/customers/3435" }},
         "total": 30.00,
         "currency": "USD",
         "status": "shipped",
         }]
      },
      "status": "PENDING",
      "startDate": "24/5/2016"
      "endDate": "24/5/2016"
}
```

- *Updating an Order using PUT*

PUT is an idempotent request that is used to update a resource. For instance, if a client knows that a specific resource resides at http://contoso.com/orders/1234, it is possible to PUT a new resource representation of this order directly through a PUT on this URL. Often times a client just needs to send the data that needs to be updated. Once the request is successful, the server responds with a 200 response.

```
- REQUEST -
PUT /orders/1234 HTTP/1.1
Host: www.contoso.com
Content-Type: application/hal+JSON

{
   "startDate": "21/04/2016",
   "endDate" : "22/05/2016",
}

- RESPONSE -
200 OK HTTP/1.1
```

- *Deleting an Order using DELETE*

A client is able to delete or deactivate an existing resource by using HTTP DELETE on the URL of an item resource. A HTTP status code of 204 is usually returned if the delete request is successful.

```
- REQUEST -
DELETE /my-collection/1 HTTP/1.1
Host: www.contoso.com

- RESPONSE -
204 No Content HTTP/1.1
```

2.5.2 SIREN

SIREN, is a hypermedia format that is normally used to represent generic entities that contain actions for modifying entities and links for client navigation. [5] Siren can be compared to a data model for ones API because a client can get insight about the data as a class e.g. When defining our response as an order class, it is possible to see the methods and data used to process actions.

- *Entities*

As described by Sookocheff [5], every SIREN entity usually has an optional class that presents the nature of an entity which defines the type of resource to be returned by the API. This property is like the data model of an API. The example below shows how entities are used in a response. Any related entities that one would require to embed in the current representation are nested as a list of entities. Each entity can have a class, properties and additional entities as seen in Appendix VII.

- *Actions*

As seen in previous Hypermedia types, the ability to dictate what requests can be made to alter the application state is one of the missing elements. SIREN defines actions that a client can take on a resource. SIREN provides actions that show the available HTTP request methods. The actions also include the URL for the request along with fields or variables that the URL accepts. Below is an example, shows a resource where listing orders can offer an action to add an order to the list, or search for an order. [5]

2.5.3 Collection +JSON

The Collection +JSON is a JSON Hypermedia type that contains a full read and write capability for simple lists or collections. It also supports the basic semantic CRUD along with custom queries with templates that are formatted like HTML "GET" forms. [4] The server appends the template to the response representation for create and update operations. In a Collection, every item contains a URL via the assigned Href property and an optional array of one or more data elements or more link elements. Both arrays have a name property in each object in the collection for domain-specific semantic information on the elements. [4] (e.g. "data": [{"name": "first-name"...}...]).

The following sections will describe how the process of reading and writing data using the Collection +JSON Hypermedia type as well as parsing and executing Query Templates will come into play. Collection +JSON is well suited for dealing with collections but it also represents a single item as a collection hence it handles most API responses elegantly. However, as a prerequisite for Collection +JSON, a response must have a collection object with a version and a URL pointing to itself.

- *Links*

The Links property can be used to as a collection property or for individual items in the collection. They may include a name and a prompt which is useful for creating reference HTML forms to reference the collection or item. In this example below we add links for the Orders picture and orders.

- *Templates*

In Collection +JSON templates are uniquely suited for handling collections. A template is used to represent an item in the collection. The client then uses it as a way to collect data so as to

POST it to the collection to create a new element or as a PUT to update an existing item. In this example below the template is used for adding more orders to the customer's list of orders.

- *Adding an Item*

In order to create a new item in the collection, a template is used to compose a valid representation. It is then sent to the server using a HTTP POST so as to create the resource. [4] For example in order to add an order to this collection, one would POST the data as defined by the template to the href link defined in the collection (http://contoso.com/orders/1234). Once the item is created successfully, the server responds with a status 201 together with a location header that has a URL of the newly created item resource. Once an item resource has been created and its URL is known, that [6]URL can be used to read, update, and delete the resource.

```
- REQUEST -
POST /my-collection/ HTTP/1.1
Host: www.contoso.com
Content-Type: application/vnd.collection+JSON
{ "template" : { "data" : [ ...] } }

- RESPONSE -
201 Created HTTP/1.1
Location: http://www.contoso.com/my-collection/1
```

- *Reading an Item*

In order to receive items in a collection one has to initiate a GET HTTP request to the URL of a collection. A Collection +JSON [4] document that contains one or more item objects in an array is returned. Clients usually retrieve an existing item resource by sending an HTTP GET request to the URL of an item resource. If the request is valid, the server will respond with a representation of that item resource. Below is an example showing how the response may describe all or partially, the items in a collection[7]. Note that the valid response is actually a complete collection document that contains only one item (and possibly related queries and template properties).

```
- REQUEST -
GET /my-collection/1 HTTP/1.1
Host: www.contoso.com
Accept: application/vnd.collection+JSON

- RESPONSE -
200 OK HTTP/1.1
Content-Type: application/vnd.collection+JSON
Content-Length: xxx

{ "collection" : { "href" : "...", "items" : [ { "href" : "...", "data" : [...] } }}
```

A typical response includes a list of items in the collection, however for a single resource contains a list of a single element. [5] The properties of each element are given by explicit name/value pairs within a data attribute as in the example below.

```
GET http://contoso.com/orders/1234
{
    "collection": {
        "version": "1.0",
        "href": "http://contoso.com/orders",
```

[6] http://amundsen.com/media-types/collection/format/#types-URL
[7] http://amundsen.com/media-types/collection/format/#objects-collection

```
        "items": [
            {
                "href": "http://contoso.com/orders/1234",
                "data": [
                      { "name": "orderId", "value": "1234", "prompt": "Identifier"
},
                      { "name": "name", "value": "Caterpillar", "prompt": "Full
Name" },
                      { "name": "alternateName", "value": "CAT", "prompt": "Alias" }
                ]
            }
          ]
    }
}
- REQUEST -
GET /my-collection/1 HTTP/1.1
Host: www.contoso.com
Accept: application/vnd.collection+JSON
- RESPONSE -
200 OK HTTP/1.1
Content-Type: application/vnd.collection+JSON
Content-Length: xxx

{ "collection" : { "href" : "...", "items" : [ { "href" : "...", "data" : [...] } ]}
```

- *Updating an Item*

A client is able to update the contents of a resource, by using the template object to create the replacement item representation to the server. Using the HTTP PUT request, the representation is sent to the server. A status code 200 is returned successfully if the update request successfully submits the data. [12]

```
- REQUEST -
PUT /my-collection/1 HTTP/1.1
Host: www.contoso.com
Content-Type: application/vnd.collection+JSON
{ "template" : { "data" : [ ...] } }

- RESPONSE -
200 OK HTTP/1.1
```

- *Deleting an Item*

A client is able to delete or deactivate an existing resource by using HTTP DELETE on the URL of an item resource. A HTTP status code of 204 is usually returned if the delete request is successful.

```
- REQUEST -
DELETE /my-collection/1 HTTP/1.1
Host: www.contoso.com

- RESPONSE -
204 No Content HTTP/1.1
```

- *Queries*

The queries property in Collection +JSON is used to define supported queries in the collection. The data object contained in a queries object is used to structure the data to be sent when the query is executed. One can create filters or search for specific data in a resource.

```
GET http://contoso.com/orders/1234

"queries": [
            {
                "rel": "search", "href": "http://contoso.com/orders/1234/search",
"prompt": "Search",
                "data": [
                    {"name": "search", "value": ""}
                ]
            }
        ],
        "template": {
            "data": [
                {"name": "orderId", "value": "", "prompt": "Identifier" },
                {"name": "name", "value": "", "prompt": "Full Name"},
                {"name": "alternateName", "value": "", "prompt": "Alias"},
                {"name": "image", "value": "", "prompt": "Picture"}
            ]
        }
    }
}
```

Using the template and queries in the response Collection +JSON makes navigation relatively simpler without needing to understand the full meaning of the API. It also provides a level of interoperability between APIs using the Collection +JSON media type since they all have a similar format.

2.5.4 Conclusion

In summary, it is clear that, Collection +JSON differentiates its capabilities from other representations by clearly illustrating the actions that can be performed against an API by the use of well described features that are well represented with its standardized responses. Below is also a summary of the characteristics of the other Hypermedia representations:

1. **JSON-LD** is great for extending existing APIs without presenting breaking changes. This extension generally serves as an approach to self-report your API. [5] If an engineer wishes to add operations to a JSON-LD response one can use HYDRA which adds the vocabulary for communicating using the JSON-LD specification.
2. **HAL** usually has light weight syntax and semantics which make it appealing in a lot of contexts. [5] It is minimalistic in terms of representation and it offers the Hypermedia benefits without adding too much complexity to the implementation. One can implement as many functions as possible to the response, unfortunately for HAL, like **JSON-LD** it lacks the support for specifying actions. We also find that it is barely possible to include more than one action per request.
3. **SIREN** attempts to represent generic classes of items to overcome HALs lack of support for actions. [5] It does this well by introducing classes to your model which brings out a sense of type information to your API responses.
4. **Collection +JSON** provides a variety of rich and well-structured features such as templates, queries, links, commands and it also represents single items as well. [5] It does a good job when representing data collections. With the ability to list supported queries for your collection templates that clients can use to alter the state of an item or a collection.

2.6 Domain Driven Design (DDD)

Domain Driven Design is a practice of developing complex systems by focusing on mapping activities, tasks, events and data in a problem domain into the software artefacts of a solution domain [13]. Domain Driven Design involves the collaboration between people who know the problem domain (domain experts) and the people who know the solution domain (design/architecture experts). [14] The shared model facilitates the understanding between the people of the different domains as they are able discuss the shared knowledge base with shared concepts through a shared language. This communication facilitates finding the actual needs within the problem domain and the most suitable solutions to meet those needs as explained by Bets [8], he points out that:

- The domain models used in DDD should capture the rich business knowledge in detail, but it should also be close to the code that is written. This is useful in that it helps the business and development team communicate with each other clearly.
- By capturing the valuable domain knowledge, in the long term Domain models are very useful if well maintained. They make it easier for future maintenance and enhancement of the system.
- DDD offers guidance on how to dissect large problem domains effectively enabling multiple teams to work in parallel, and also enable you to focus the most appropriate resources on critical parts of the system that offer the greatest business value.

2.6.1 Domain Model

The domain model is what captures the valuable or unique essence of the business. DDD at its core, encompasses the concepts of the domain model which are designed and built by both the business domain experts and the software developers as explained by Dominic. [8] DDD focuses on the process of creating, maintaining, and using domain models. Domain models are composed of elements like aggregates, entities and value objects. [8] Eric Evans [13] explains that domain models are usually described using terms from a ubiquitous language. The domain model serves several functions:

- It captures the relevant domain knowledge from the domain experts.
- It enables the team to identify the scope and verify the consistency of that knowledge.
- The model of the domain is expressed in code by the developers.
- It reflects evolutionary changes in the domain.

Below are the characteristics of a rich domain model.

- A rich domain focuses on the business model, strategies and business processes of a specific operational business domain.
- It should be isolated from other domains in the business as well as other layers in the application architecture and should be loosely coupled with no dependencies on the layers on either side of domain layer or other layers in the application (i.e. database and facade layers).
- It should be reusable to avoid redundancy and duplicate models of the same core business domain elements.
- It should be an abstract and cleanly separated layer enabling easier maintenance, testing, and versioning with its domain classes with the possibility of having unit tests outside the container and from inside the IDE (Integrated development Environment).
- It should be designed using a POJO (plain old java object) programming model without any technology or framework dependencies.
- It should be independent of persistence implementation details.

- It should outlive any infrastructure frameworks by having minimal dependencies on any external frameworks.

2.6.2 Ubiquitous language

Ubiquitous language, as termed by Eric Evans, is used for building up a common, rigorous language between developers and users in Domain Driven Design. Evans clarifies the importance of testing ubiquitous language in conversations between domain experts, because it is what makes the domain model. He also stresses that the language and the domain model should evolve as the overall team's understanding of the domain grows. The risk of confusion is reduced when they use similar terms for objects and actions within the domain like conference, employee and reservation.

2.6.3 Domain Model Elements

A complete domain model comprises of different elements so as to fully make sense in a domain. Eric Evans [13] uses the following terms to describe the common building blocks that make up a domain model in DDD.

- **Entity.** Entities in most cases take up the identity of a real life object, and are defined by their identity which continues through time. A good example, is in Contoso where it remains unique in the system, while many of its attributes like its name and size could change over time in the system. The object that represents the conference does not always exist in the system's memory it is also persisted in the database and is only re-instantiated when required.
- **Value object**. A Value object is usually immutable where not all objects are defined by their identity. For some, the values of their attributes are what is important. A good example is in Contoso where there is no need to set the identity of every attendee's address. The reason is that the attendees may need to share the same address from an organization e.g. city, street, state etc.
- **Service**. A service is used to process data and transform the state of an entity. Unlike an entity or a value object, a service is stateless. For example, in Contoso we may implement an invoicing system as a service where the system automatically processes invoices after it receives a result once a payment is successfully processed by a payment processing service.

2.6.4 Bounded context

A bounded context usually represents a section of the system that has its boundaries clearly defined. In complex systems, it becomes very cumbersome to maintain a single domain model because of its size and the effort that it would take to manage its complexity in a coherent and consistent manner. DDD introduces the concepts of multiple models and bounded contexts to manage such complex scenarios. [8] Dominic [8] explains that in a system, one may choose to make use of smaller multiple models where each model focuses on a single specific aspect that groups the functionality within the overall system. Every model has its own ubiquitous language of the domain. This facilitates a clear separation from other bounded contexts within the system. [7] Bounded contexts can encompass everything from a data store to the UI which can similarly apply to multiple bounded contexts. A good example would be where an attendee in Contoso exists in other bounded contexts such hotel reservations and sauna reservations. The domain expert, would attribute different behaviours for the different versions of the attendee in the other contexts. For example, in the hotel reservations bounded context, the attendee is associated with a registrant who makes the bookings and payments. The Information of the registrant becomes

irrelevant in the context of the hotel reservations, where smoking preferences are important. The advantage of splitting a complex domain into multiple contexts is that one can use different implementation architectures in the different contexts. For example, we can have a bounded context that uses a DDD layered architecture or one that uses a two-tier CRUD. [8] Figure 6 shows a complex system that comprises multiple bounded contexts that use different architectural styles.

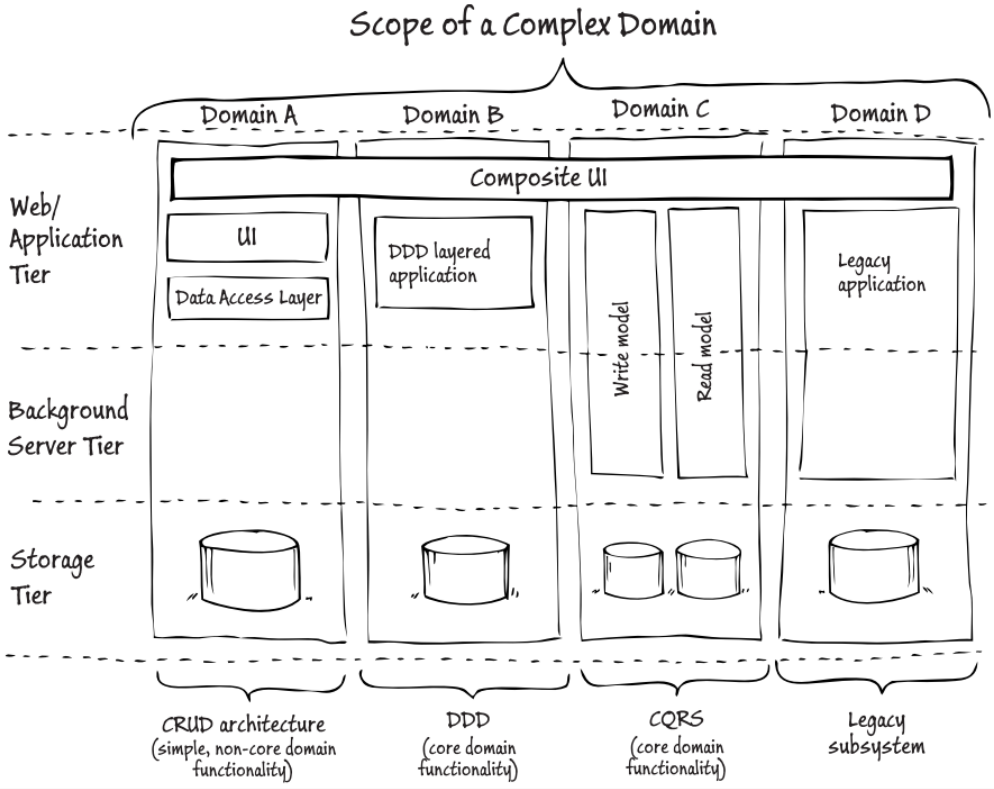


Figure 6 scope of a complex system with multiple bounded contexts

2.6.5 Enterprise Application Architecture

A typical enterprise application architecture is made up of four conceptual layers as seen in the figure below:

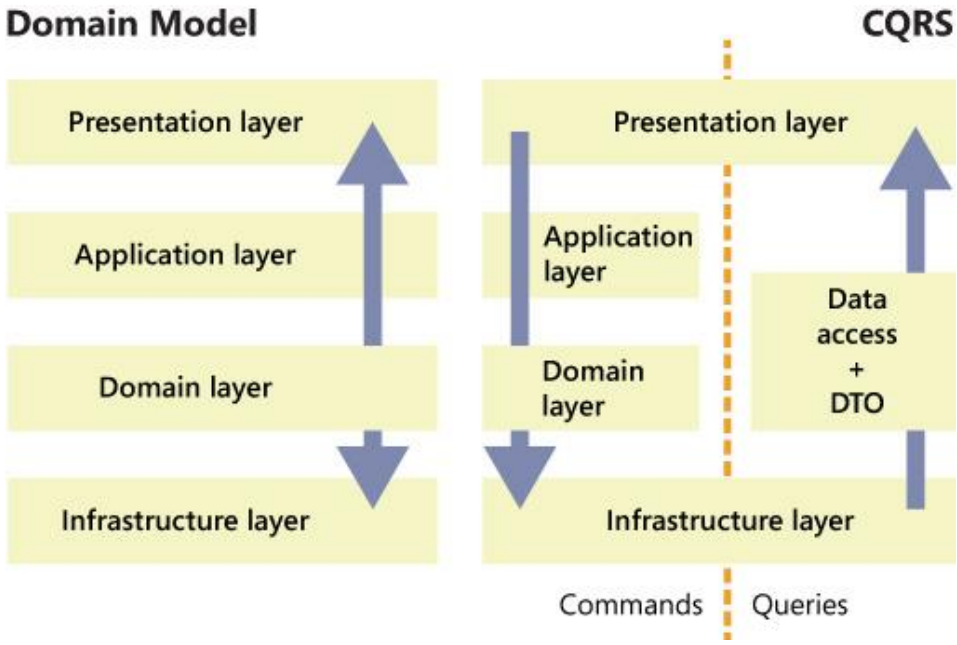


Figure 7 Comparison of a Domain Model and CQRS (adopted from [15])

- **Presentation Layer:**
  The User interface is responsible for facilitating interaction with the application by presenting information to the user and interpreting user input commands via peripheral devices in to actions and results. A combination of the user interface and the controller is what mediates user interaction converting it to commands for the model or view.

- **Application Layer:**
  This layer coordinates application activity by hosting the state of an application tasks progress. It usually does not contain any business logic nor does it hold the state of business objects, but it can hold the state of an application task's progress.

- **Domain Layer:**
  The domain layer usually contains information about the business domain and the state of business objects. Persistence of business objects and possibly their state is delegated to the infrastructure layer.

- **Infrastructure Layer:**
  This layer is used to support other layers by providing communication between layers, implementing persistence for business objects and containing supporting libraries for the user interface layer etc.

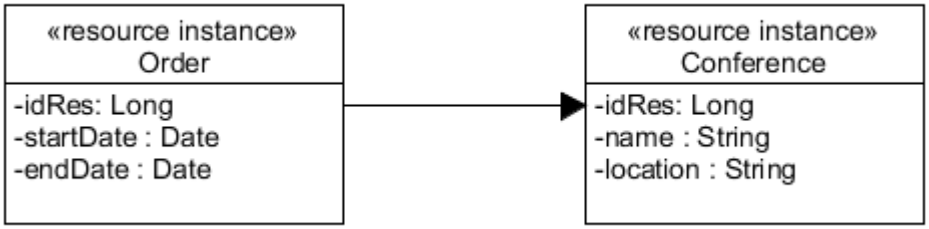
The Application layer

The application layer caters for the navigation between the UI screens as well as the interaction between the application layers of other systems. It is also used to perform basic validation on user input data before transmitting it to the other lower layers of the application. [14] It does not usually have any business, domain related or data access logic. It also does not contain any state reflecting a business use case but it can be used to manage the state of the user session or the progress of a task.

- Resources

A resource is what is returned when a client makes a request to a server. In modern applications, it represents the state of the domain object as stored in the System. Using similar variable names one can model the kind of information that should be returned to the client without actually altering the data in the database. For Example, an Order would be fetched as an Order Resource. Typically a Resource is used to vary the amount of information that can be exposed to client. It is possible to expose other Hypermedia embedded resources such as calculated values, links, and formatted dates etc.

A resource has at least a URL e.g. ***http://www.contoso.com/orders/1234***



- Controllers

It is in the application layer that we find the Controllers that are used to handle the HTTP requests that are received by the application from the client. It is responsible for generating a response that can be understood by the client using the CRUD methods.

- Assemblers

It is used by a controller to assemble a resource together by fetching information from the domain object and setting the variables of the Resource.

The domain layer

As described by Srini, Domain objects epitomizes the state and behaviour of business entities. [14] Cases of business elements in the Conference management system are Order, Conference, and Registrant. The domain layer is in charge of the ideas of the business domain, data about the business use case and its business rules. It can likewise deal with the state (session) of a business use case if the use case spans multiple user requests (e.g. seat registration process that consists of many steps: registrant entering the conference details, system returning the available seats based on the type and class, user reserving a specific seat, and finally the system booking the seat for that user). It contains service objects, which are not part of any domain object, that only have a defined operational behaviour that operates as the heart of the business application and should be well isolated from other layers of the application. It is also worth noting that the domain layer is also not dependent on the application frameworks used in other layers. (I.e. Hibernate)

The following design aspects are considered as the main ingredients of the current DDD implementation recipe:

i. Object Oriented Programming (OOP)

OOP is usually one the most vital anchors in the process of domain implementation. By applying OOP concepts like encapsulation, inheritance, and polymorphism objects are normally represented by the use of Plain Java Classes and Interfaces. Most domain elements are true objects that have both State (attributes) and behaviour (methods or operations that change the state of domain) which compare to real world concepts and can fit right in with OOP concepts. Entities and Value Objects in DDD are great cases of OOP ideas since they have both state and behaviour. [14]

In a typical Unit of Work (UOW), domain objects need to collaborate with other objects whether they are Services, Repositories or Factories. Domain objects also manage other important concerns like transaction management, caching, domain state change tracking, auditing which are cross-cutting in nature. These reusable non-domain related concerns are usually in most situations scattered and duplicated throughout the code including the domain layer. The end result of having this logic in the domain objects is what leads to a tangled and cluttered domain layer with non-domain related code. Unfortunately OOP alone cannot provide an elegant design solution for DDD, because of its weaknesses in when it comes to managing the code dependencies without tight-coupling between objects and isolating cross-cutting concerns. This is where design concepts like Dependency Injection and Aspect Oriented Programming can be used to hand in hand with OOP to enhance modularity , reduce tight coupling and to better manage the cross-cutting concerns.

ii. Dependency Injection(DI)

DI is an effective configuration technique that moves configuration and dependency code out of the domain objects. DI becomes a “must have” in DDD implementation because of the design dependency of domain classes on Data Access Object (DAO) classes and service classes. DI injects other objects like Repositories and Services into Domain Objects which further facilitates a less cluttered and loosely coupled design as presented by Srini in [14] a good example in the implementation, is injecting a service class like PurchaseOrderService in the controller class initialization so as to access the domain methods like In the sample implementation. Srini further explains that entities reference Repositories via DI.

iii. Aspect Oriented Programming(AOP)

Srini [14] explains in his research that AOP helps in improving a design and removing clutter by removing the cross-cutting concerns like auditing and domain state change tracking among other issues from the domain objects. It can be used to inject collaborating objects and services into domain objects especially the objects that are not instantiated by the container like persistence objects. Other aspects in the domain layer that could use AOP are caching, transaction management and role based security authorization.

DDD cannot be implemented without help of AOP and DI as mentioned by Ramnivas Laddad [16]. In the presentation, Ramnivas discusses the concept of "fine grained DI" using AOP to make domain objects regain smart behaviour. He mentioned that in order to provide rich behaviour, domain objects need to access other fine grained objects by injecting Services, Factories, or Repositories into Domain Objects which inject dependency at constructor or setter invocation time. This technique also eliminates the "Big Fat Service" anti-pattern which is the result of coupling, tangling and scattering of the application code. [14]

iv. Annotations

According to Srini [14], Annotations are used to minimize the required artefacts for implementing remote services such as Web Services. They are also used to define and manage Aspects and DI by simplifying the configuration management tasks. Spring[8], Hibernate[9] and other frameworks make use of Annotations to configure components in the different layers of a Java[10] enterprise application. Annotations are useful in that they can be used to generate the boiler plate code. Annotations should not mislead the understanding of the actual code e.g. In Hibernate Object-relational mapping (ORM), one specifies the SQL table or column name right next to the attribute name. Some of the annotations are used in the models of the proof of concepts, i.e. Entity objects like Order, Purchase Order, and Conference. These objects use @Entity annotation to wire the Repository objects. And the Service classes use @Service annotation to automatically initialize the service wherever it is injected using the @Autowired annotation.

2.7 Command Query Responsibility Segregation (CQRS)

CQRS is an acronym for Command Query Responsibility Segregation (CQRS), which is a development technique proposed by Greg Young [8] that isolates queries and commands of a framework in to two separate subsystems.

2.7.1 Introduction

In his book "Object Oriented Software Construction," Betrand Meyer [17] introduced the term Command Query Separation to describe the principle that an object's methods should be either commands or queries. He went further to explain that a query returns data and does not alter the state of the object; while a command changes the state of an object but does not return any data. The benefit is that you have a better understanding what does, and what does not, change the state in your system. CQRS takes this principle a step further to define a simple pattern. In summary a command is any method that mutates state and a query is any method that returns a value as explained by Greg Young [8].

2.7.2 Read and write Sides

In REST, GET requests wire up to fetching of date as opposed to what PUT, POST, and DELETE requests wire up to. This simple pattern enables a developer to meet a wide range of architectural challenges, such as scalability, complexity, and the constantly changing business rules in some portions of your system [8].

[8] https://spring.io/ is a framework used to develop web applications
[9] http://hibernate.org/ is a framework used to resolve database driven commands
[10] https://www.java.com/en/

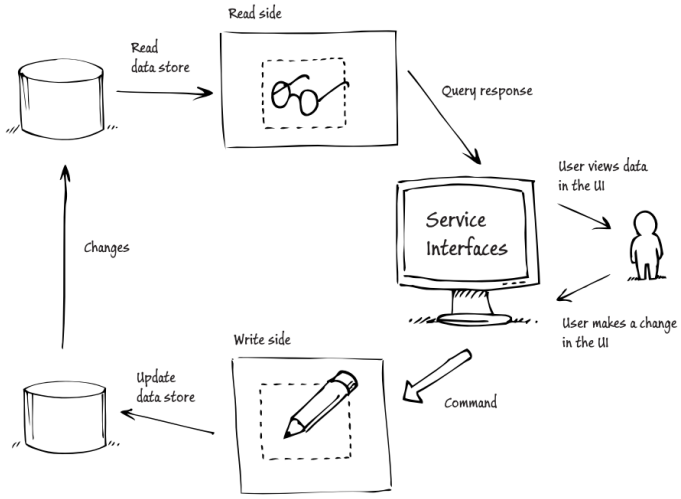


Figure 8 CQRS Model diagram (adopted from [8])

In order to apply CQRS, one should do so in a specific bounded context, as opposed to the whole system level. Figure 8 shows a typical CQRS application where the read and write sides are separated.

In summary, CQRS pattern applies in the different portions of a system that one wishes to implement independently where there are clear business benefits.

2.7.3 Conclusion

Exposing a CQRS service through a REST API is not only possible but the richness of HTTP semantics allows for a fluent and efficient API to be built on top. This process involves building a public domain composed of commands, queries (input/output messages) and resources that are concurrency and caching aware. Also, we need to map internal domain's queries and commands to HTTP verbs and use status codes to convey state transitions and exceptions. Use of 5LMT helps with building resources that are fully REST-ful and not RPC-styled.

2.8 Problem

Our problem scenario discusses a customer who creates an order that contains four Regular seats. He wishes to upgrade two of the seats from Regular to VIP seats. However before showcasing the issue at hand, we must start by creating an order using the post method, by sending a combination of the customer's data and the selected seats. The JSON data that is sent for creation of an order contains the customer and a collection of seats which comprise of the seat number, the start date, end date and the row letter as seen in Figure 9.

```
POST /orders/
{
 "user" : { "href": "/users/1234" },
 "seats": [{ "seatNumber": 310,"row": "A",
"startDate": "11/10/2016","endDate": "21/10/2016"},
           { "seatNumber": 311,"row": "A" ,
"startDate": "11/10/2016","endDate": "21/10/2016"},
           { "seatNumber": 312,"row": "A",
"startDate": "11/10/2016","endDate": "21/10/2016" },
           { "seatNumber": 313,"row": "A" ,
"startDate": "11/10/2016","endDate": "21/10/2016"}]
}
```

Figure 9 JSON object to create an order using POST operation

Once the Order is created successfully, a 201 response is returned by the server with a JSON response that contains the generated HREF of the order and a representation of what has been stored in the system as seen below in Figure 10. Additional information like the status of the seat, e.g. Booked and seat type e.g. regular is returned.

```
Status: 201 Accepted
Location: http://contoso.com/orders/12345

{
"_links": {
  "self": {
   "href": "/orders/12345"
  }
 },
 "user" : { "href": "/users/1234" },
 "seats": [
  {
   "_links": { "self": { "href": "/orders/12345/seats/1"}}, "seatNumber": 310,
   "row": "A","seatType":"REGULAR", "status": "BOOKED", "startDate":
"11/10/2016","endDate": "21/10/2016" },
   "_links": { "self": { "href": "/orders/12345/seats/2"}}, "seatNumber": 311,
   "row": "A","seatType": "REGULAR", "status": "BOOKED", "startDate":
"11/10/2016","endDate": "21/10/2016" },
   "_links": { "self": { "href": "/orders/12345/seats/3"}}, "seatNumber": 312,
   "row": "A", "seatType": "REGULAR", "status": "BOOKED","startDate":
"11/10/2016","endDate": "21/10/2016" },
   "_links": { "self": { "href": "/orders/12345/seats/4"}}, "seatNumber": 313,
   "row": "A", "seatType": "REGULAR", "status": "BOOKED", "startDate":
"11/10/2016","endDate": "21/10/2016" }]
}
```

Figure 10 JSON response of a created Order

In order to update the existing seats, the customer has to make a request using the patch operation by sending the intended seat changes with a reference to the order id.

Below in Figure 11. We see a user attempt to update the seat by selecting a different seat number and a different row letter.

```
PATCH /orders/12345

{
 "seats": [
  {
   "href": " /seats/1", "seatNumber": 210,  "row": "P" },
   "href": " /seats/2", "seatNumber": 211,  "row": "P" }]
}
```

Figure 11 Update using PATCH request

In one scenario, once the customer submits the updated details, the system responds with a 202 Accepted response as seen in figure 12 and a JSON object that contains all newly updated details. However, in our scenario, only one seat is updated from Regular to VIP while the other one is left unchanged, because the seat is either unavailable or already booked by another customer.

```
Status: 202 Accepted
Location: http://contoso.com/orders/12345

{
"_links": {
  "self": {
   "href": "/orders/12345"
  }
 },
 "user" : { "href": "/users/1234" },
 "seats": [
  {
   "_links": { "self": { "href": "/orders/12345/seats/1"}}, "seatNumber": 310,
   "row": "A","seatType":"REGULAR", "status": "BOOKED", "startDate":
"11/10/2016","endDate": "21/10/2016" },
   "_links": { "self": { "href": "/orders/12345/seats/2"}},  "seatNumber": 211,
   "row": "P", "seatType": "VIP", "status": "BOOKED" , "startDate":
"11/10/2016","endDate": "21/10/2016"},
   "_links": { "self": { "href": "/orders/12345/seats/3"}}, "seatNumber": 312,
   "row": "A", "seatType": "REGULAR", "status": "BOOKED" , "startDate":
"11/10/2016","endDate": "21/10/2016"},
   "_links": { "self": { "href": "/orders/12345/seats/4"}}, "seatNumber": 313,
   "row": "A", "seatType": "REGULAR", "status": "BOOKED", "startDate":
"11/10/2016","endDate": "21/10/2016" }]
}
```

Figure 12 Update using PATCH response

It is common in applications today to have the backend implementation, compare the update parameters with the existing stored data based on various rules in order to determine which parameters to update. E.g. if the Seat number and row letter are updated, the system would check to see if it is already assigned to anyone, and then update the customer's selection accordingly. Often times the implementation is very complex.

In another scenario, the customer changes the duration of the booked seats by extending and reducing the duration of one of the seats. The customer does this by only changing the start date of one seat, and the end date of another as seen in Figure 13.

```
PATCH /orders/12345

{
 "seats":
   [{
   "href": " /seats/1", "startDate": "21/11/2016"},
   "href": " /seats/2", "endDate": "16/11/2016"}]
}
```

Figure 13 Update seat Start Date and end Date using PATCH request

In order for the system to determine what part of the domain to update, similar to the updating of seats, when the customer extends the duration of the seats, the system successfully updates the data and returns a 202 Accepted response as seen in Figure 14. The same complex issue is highlighted in other Hypermedia formats such as HYDRA, SIREN, and JSON-LD, where a complex algorithm is implemented to selectively update the correct information, hence they are not well suited for rich update operations because the results of the update operations are not transparent.

```
Status: 202 Accepted
Location: http://contoso.com/orders/12345

{
"_links": {
  "self": {
   "href": "/orders/12345"
  }
 },
 "user" : { "href": "/users/1234" },
 "seats": [
  {
   "_links": { "self": { "href": "/orders/12345/seats/1"}}, "seatNumber": 310,
   "row": "A","seatType":"REGULAR", "status": "BOOKED", "startDate":
"11/10/2016","endDate": "16/10/2016" },
   "_links": { "self": { "href": "/orders/12345/seats/2"}}, "seatNumber": 211,
   "row": "P", "seatType": "VIP", "status": "BOOKED" , "startDate":
"10/10/2016","endDate": "21/10/2016"},
   "_links": { "self": { "href": "/orders/12345/seats/3"}}, "seatNumber": 312,
   "row": "A", "seatType": "REGULAR", "status": "BOOKED" , "startDate":
"11/10/2016","endDate": "21/10/2016"},
   "_links": { "self": { "href": "/orders/12345/seats/4"}}, "seatNumber": 313,
   "row": "A", "seatType": "REGULAR", "status": "BOOKED", "startDate":
"11/10/2016","endDate": "21/10/2016" }]
}
```

Figure 14 JSON response for PATCH Update

2.8.1 Implications on both the server and client side

In order for the application to receive and process HTTP requests, a controller is used to process and handle the creation of resources that respond to requests from clients. We find that developers tend to create applications that are RPC-styled as seen in Figure 9, with individual methods to process each URL request as seen in the example below.

We find that in some applications the code that should be in the domain layer creeps into the presentation layer and the same methods are reused across the entire API in other controllers. This becomes tedious to maintain in that one has to upgrade the client's user interface to match up with every URL in the API. It also causes breaks in the application if one changes any of the code in the controller such as the name of the URL.

An example is when a developer makes the change from /rest/orders to rest/purchaseorders in the client's code and breaks the application. One has to modify the clients in order to have the orders URL working as it should. In fact, when one person changes the variables of the resource, it is likely that they have to update multiple CRUD interfaces to match up with the newly updated variable names.

In this section, we look at the various alternative approaches that are currently used in applications today. We will describe how to update the orders and the various implications on the server side depending on the approach used to update the orders. Overall we see that this breaks away from the true CRUD operations.

i. Adding a verb as the URL for the operations.

In this approach, one would append a verb to the update URL as follows

PUT orders/1234/updateseatposition.

A full representation similar to the one used in the Patch Update operations in figure 4 would be sent with the update details. Once the request is sent, it is expected that the server side will process the data and return a response similar to figure 4.

In the event that one would want to add another operation, additional code would be added explicitly in the server and client side to process the received JSON resource and generate the JSON response based on the user's input of the data respectively. This presents various problems such as the vulnerability of the API, where the client side can be broken when the API is changed. Therefore we find that there are redundant development processes that must be adhered to in order to maintain the integrity of the application. We also find that we would have multiple operations using the PUT request method thus it is no longer a true CRUD system. The advantages of this format are that it is easy to know what the specific operation does, but on the other hand, it is redundant because it is already mapped in the domain layer.

We also find that one would have to explicitly write code to process the CRUD of the JSON data received from the backend every time as seen in figure 15.

ii. **Object comparison via regular Update request**.

In this approach, one would update any part of the order using a PATCH or PUT request method via a URL e.g. PUT orders/1234. A customer would send an update request similar to figure 4, figure 6 or figure 8 to the server and the system would fetch the existing order from the database. The server side would then selectively update the resources between both existing and updated information. The Server would then compare the JSON representation against the existing data

and execute required commands. E.g. if in the order the Seat Number and Seat row were changed, the system would check to see if the desired seat is booked or available. It would then only update the seat where necessary. This approach comes with added complexities as one would have to write a very complex algorithm to compare and selectively update the data accordingly.

iii. **Adding a HEADER to the request as a hint.**

In order to complement the intended behaviour, the header is passed with the PUT update request to explicitly perform an operation. E.g. PUT orders/1234 with a header parameter e.g. Action: updateseatposition. The implementation should then analyze the payload and choose how to update the data accordingly, by comparing the part of the request that is different from the data in the database. In some cases, one would have the method that updates the seat position in the domain layer while in others would have it in the controller.

3 RELATED WORK

In this chapter we discuss work related to the development of Hypermedia APIs and their milestones. From the inception of REST, there has been significant improvements and research on hypermedia formats. This section discusses other technology designs and implementations that are related to the creating a more robust API.

3.1 Collection Document

Irakli [18] created Collection Document[11] using the PHP [12](Hypertext Pre-processor) based extension of Collection +JSON. It was developed to improve on existing content management systems because of the complexity and mismatch of data communication between Content Management systems and other systems.

Collection Document is a recursive media type that combines the idea of a document and a collection to allow the recursive collection of other documents so as to describe complex domains and it was created to help publishers store, exchange and explore in a wide variety of destinations [18].

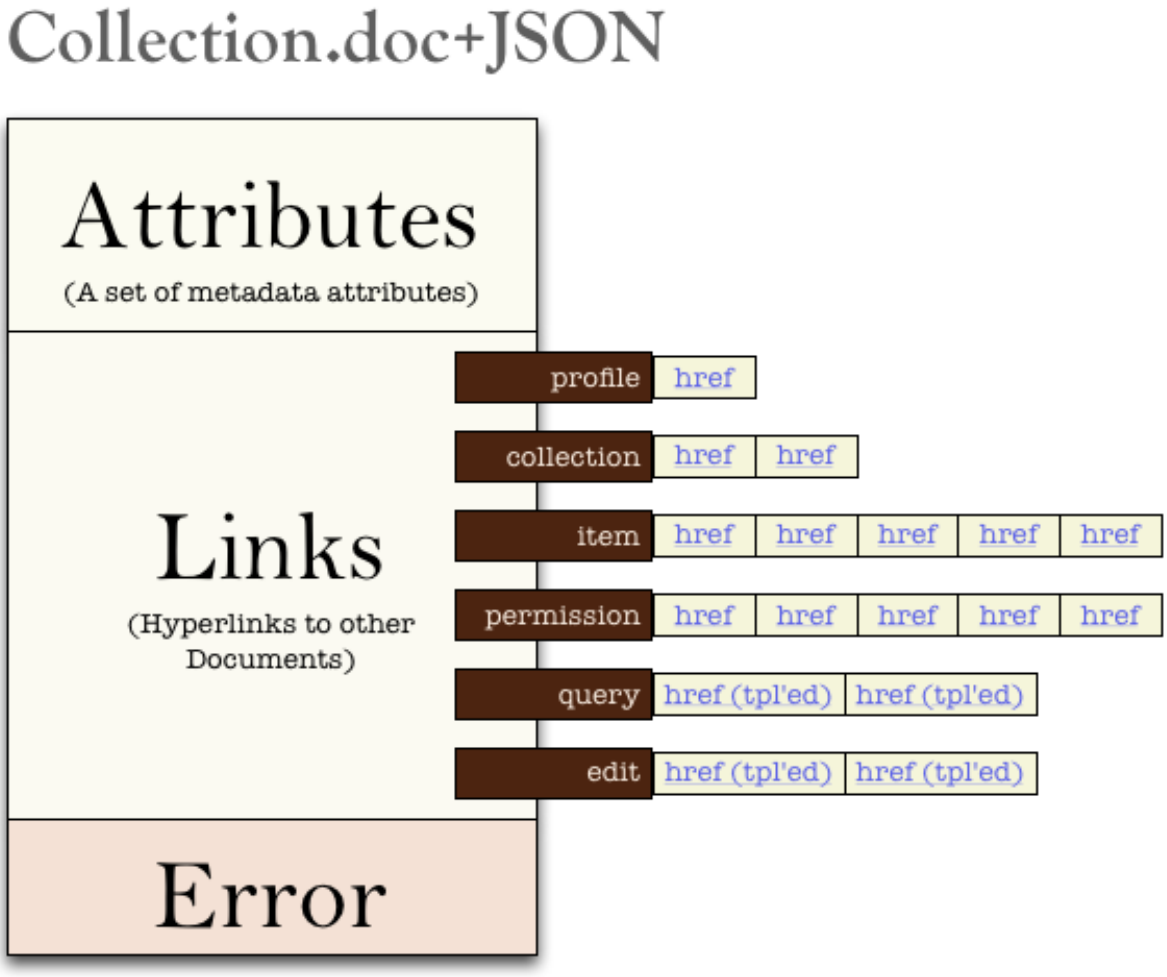


Figure 15 Collection Document structure (adopted from [18])

[11] https://github.com/publicmediaplatform/pmpdocs/wiki/Collection.doc-JSON-Media-Type
[12] http://php.net/manual/en/intro-whatis.php

Collection document also comprises of three top-level elements which are attributes, links and errors as seen in figure 15. The most important part is the links since it communicates the behaviour and relationships of the content [18]. Collection Document employs different link relation types as discussed by Irakli in his research [18]. Below is a summary of the different types:

a) Profile: Profiles are used to define additional semantics on top of the media type as used in content publishing. Profiles are usually inheritable thus facilitating re-use and collaboration. Profiles are saved just like any other document thus one doesn't have to go through a standards body.
b) Creating lists or buckets is inherited as it is one of the most important task in content management. The media type provides two different ways to do it.
c) Item link: using to top-down approach, it is used to point to other documents that it contains. The item link's relationship is a 'contains' relationship, which makes it suitable for a scenario like blog that contains blog posts.
d) Collection link: using a bottom-up approach, child documents point to the parent documents that they're associated with which is fundamentally useful and suitable for archives or topics.
e) Permission links are used to define the parameters of accessing and modifying the documents. They also point to a type of Collection Document that contains such information.
f) Query links: They are URLs with parameters that one can run to explore and filter out data.
g) Edit links are the form templates that a user can submit to when modifying data.

Collection document is defined on top of JSON because of its simplicity and popularity. The media type itself is also easily portable to XML and other micro formats extension of HTML5 [18]. Collection Document also has its primary keys and relationships in a URL-based format, which makes it possible to leverage a lot of caching, built-in security and routing capabilities of the HTTP protocol.

Hypermedia and the use of URLs for referencing is one of the ingredients that breaks the biggest constraints of traditional APIs which is the silo-ing of the content. In traditional APIs, documents need to live within the API itself for example: unfortunately a Twitter API cannot make sense of a tweet that is stored in another API. However the public media platform (PMP[13]) hypermedia format is able to make sense of data in other APIs, because it is composed of URLs to certain types of documents. PMP is a good example of an API that can have its core data existing elsewhere as long as it has a reference. When Hypermedia is used properly one ends up with a true web of APIs which is truly versatile, distributed and robust.

[13] https://support.pmp.io/docs

4 CONTRIBUTION

In Summary, the contribution of this research is advancing Amundsen's[14] Collection +JSON hypermedia API to facilitate accessing the update methods in the domain dynamically both in the API and the client. This section provides more detailed information on the techniques and approaches used to develop the solution.

We will first discuss on a high level the advancements that were made on Collection +JSON and how it improves the overall usability of the API in exposing the domain methods dynamically. We will then discuss in detail the techniques and components that were implemented to expose the domain in the API. In conclusion, we will discuss the improvements that we made on the client application to make it read and generate the respective controls needed to execute the commands. In order to demonstrate the solution, the proof of concept application was developed in JAVA Spring MVC because it is easier to add libraries with little configuration.

Below is a summary of the steps that were taken to implement the solution:

1. Implementing the Collection +JSON Library.
   Collection +JSON is open source and is implemented into the Java project by importing the library dependencies in the pom.xml file. Once the library is imported into the application, Maven automatically resolves the dependencies and downloads the JAR files that contain the Collection +JSON code. Below is a snippet of the pom.xml configuration.

```
<dependency>
        <groupId>net.hamnaberg.rest</groupId>
        <artifactId>JSON-collection</artifactId>
        <version>5.0.0</version>
</dependency>
```

2. Implementing a Generic abstract controller that implements all the basic CRUD HTTP methods.
3. Implementing a single HTTP method that handles all PUT update requests using the domain header. By removing any RPC styled methods, we can access the commands in the domain layer by name through the controller. We do this by passing the domain-header of a method during a put/post request together with the resource being processed.
4. Using DDD concepts i.e. Annotations, Dependency Injection and AOP to further simplify the resource assembler. We use annotations in the resource class to configure the properties of the variables such as name, access level, queries and commands. We also use annotations to set the access level of certain variables.

[14] http://amundsen.com/media-types/collection/format/

4.1 Advanced Collection +JSON

In order to generate a complete Collection +JSON payload, we need to implement the various blocks that encompass the complete API which comprises of different elements i.e. items, links, templates, commands and queries among others. Below in figure 16 is a high level diagram that shows the interaction between the server and the client, and how the API presents resources. The API supplies a template for create and update operations as part of the response representation. Unfortunately, it does not support multiple update operations against a resource.

In chapter 2, we discuss the implications on the client and server side, and how developers use different techniques to expose various update methods. For example, with the scenario of cancelling a purchase order, one would have to implement additional code in the API and in the client side to properly update the purchase order. One would also have to manually create the controls needed to manipulate the data such as fields, buttons and form layout.

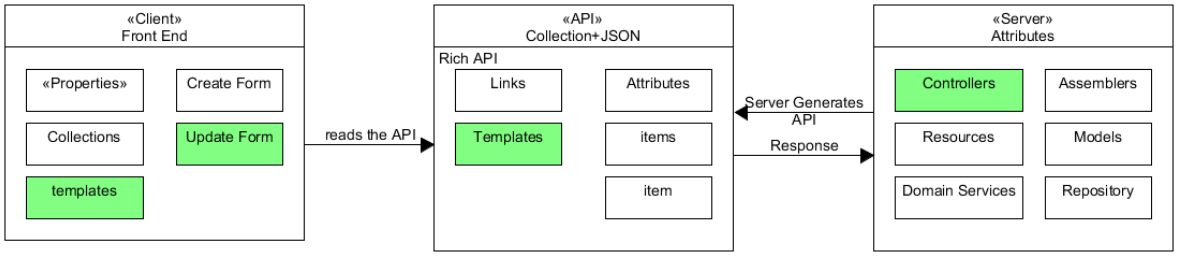


Figure 16 Basic Collection + JSON Attributes

However, we can solve this by dynamically iterating the update methods in the service layer and presenting them in the API as Commands with their own data templates. In addition to that the client should also dynamically generate the corresponding controls with elements that are based on the templates as seen below in figure 17.

Advantages:

a) *Single update URL:*
   - All the methods will be accessible through the same URL i.e. PUT /purchaseorders/2
b) *Use of a single HTTP verb "PUT":*
   - Using a single update URL means that the API conforms to RESTs principle of true CRUD because it eliminates having custom verbs and redundant methods sharing the same verb e.g. POST /purchaseorders/2 and POST /purchaseorders/2/approvePO
c) *Supports multiple commands:*
   - This means that one can dynamically add or remove as many commands as possible when required. This also facilitates having multidimensional actions within embedded resources e.g. having additional commands within a resource that is embedded inside another resource.
d) *Represent State lifecycle diagram easily:*
   - Using commands one can easily expose the appropriate actions needed to change the state of the resource into another state. A good example is providing the choice between approvePo and rejectPo after a purchase order is created.

Roy fielding infers in his research that one should include links into the resource representation to allow a person or a program to make choices and select the actions they would like to change the state of the resource. [1]

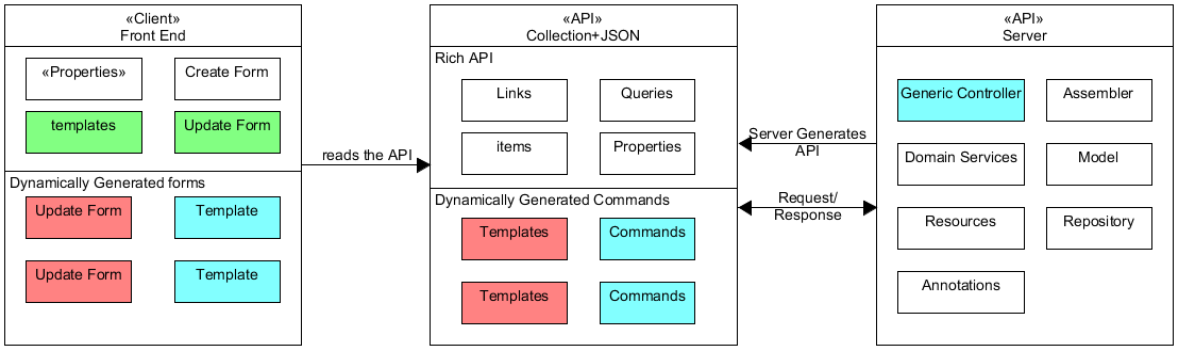


Figure 17 Advanced Collection + JSON with dynamic

Figure 18 is a good example of an API response that consists of commands and their templates. Following the resource lifecycle of creating a purchase order as shown in figure 3, we can see the three commands, i.e. requestPOUpdate, sendInvoice and closePurchaseOrder, which are exposed when the purchase order is in the Approved state. All commands share the same update URL i.e. PUT *http://contoso.com/orders/2*. The other parameters i.e. name, prompt, method and rel are used in the client to generate the form elements that will be used to collect data and submit a request. For example in order to successfully process the requestPOUpdate command which is used to request for a purchase order update, the user fills in the startDate and the endDate in the clients update form and then clicks on submit. The request sends the JSON data to the server together with the domain-header that contains the name of the method.

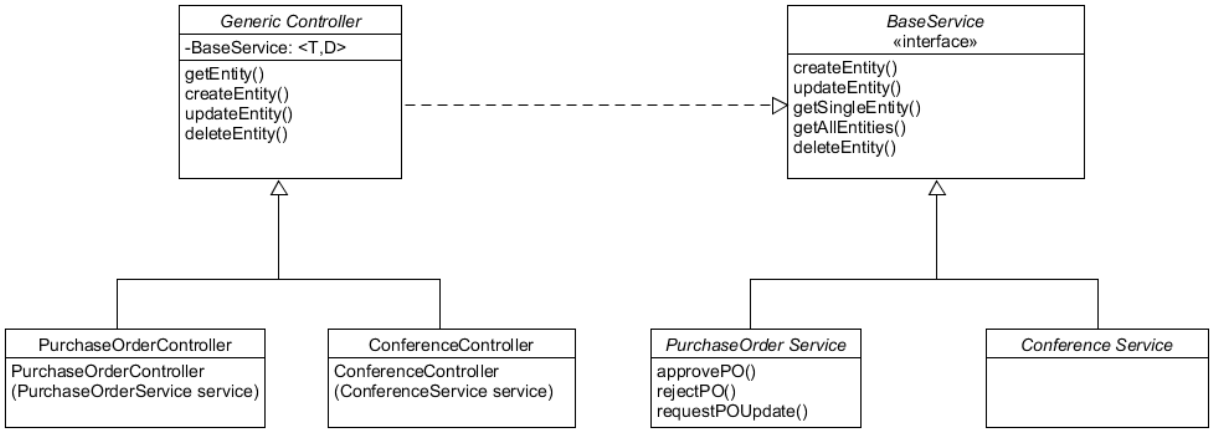
```
PUT http://contoso.com/orders/2

"commands": [
  {
         "href": "http://localhost:8686/api/pos/2",
         "rel": "requestPOUpdate",
         "prompt": "Update PO",
         "name": "Update",
         "method": "put",
         "data": [
           {
                  "name": "startDate",
                  "prompt": "Start Date",
                  "value": ""
           },
           {
                  "name": "endDate",
                  "prompt": "End Date",
                  "value": ""
           }]
  },{
         "href": "http://localhost:8686/api/pos/2",
         "rel": "sendInvoice",
         "prompt": "Send Invoice",
         "name": "Invoice",
         "method": "put"
  },{
         "href": "http://localhost:8686/api/pos/2",
         "rel": "closePurchaseOrder",
         "prompt": "Close PO",
         "name": "Close",
         "method": "put"
  }
]
```

Figure 18 Collection + JSON API of Purchase orders

4.2 The Generic Controller

Firstly the generic controller serves the purpose of eliminating redundant code that is present in controllers. Using the principles of inheritance, the abstract generic controller, is inherited by all controllers. As seen below, the controllers inherit the basic Http methods from the Generic controller.



The generic controller implements the basic CRUD Http methods and a generic interface that is implemented by all domain services. It is more viable to inherit and reuse the Http methods as opposed to rewriting them in every controller throughout the entire application as seen in Appendix d. We also accept all Http requests by using the asterisk {*} in the value of the request mapping to accept all request with a similar format.

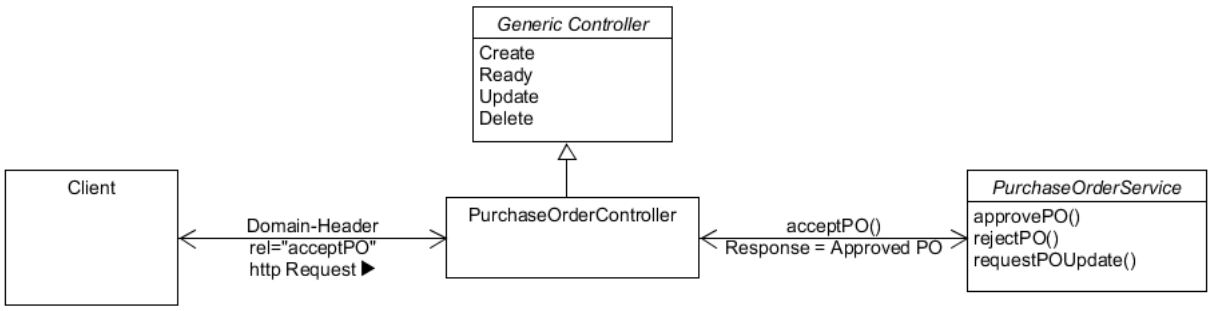
```
@RequestMapping(method = RequestMethod.PUT, value = { "/{id}/*", "*","/{id}" })
public D updateEntity(@PathVariable T id, @RequestBody D collection,
@RequestHeader("domain-model") String command) throws Exception {
 . . .
}
```

The CRUD Http methods act as singular gateways to the domain service layer, by iterating the domain service methods and selecting the desired method by name, a technique promoted by CQRS. This technique reduces the amount of code required in the controller as seen below in comparison to appendix III. We are able to initialize it with a single constructor.

```
@RestController
@RequestMapping("/api/pos")
@CJLink(name = "PO", prompt = "Purchase Orders", rel = "POs", href = "/api/pos")
public class PurchaseOrderController extends GenericController<Long, String> {

        @Autowired
        public PurchaseOrderController(PurchaseOrderService service) {
                super(service);
        }
}
```

In more detail, this is made possible using the commands. A command in the API has the name of the method assigned to the “rel” parameter. Every time a user executes the command action, the client reads the “rel” tag from the API and appends it to the JSON payload as a domain-header with the name “domain-model” as seen in Appendix V.



When the server receives the request, it uses the domain-header to select the method that matches the name e.g. in figure 18, for the command requestPOUpdate, the server fetches the requestPOUpdate method and passes the data parameters specified in its data template. Once it successfully processes the data, a response is sent back to the client.
The advantages of using this technique is that it save a lot of time on a much bigger scale with larger applications that have millions of lines of code, it reduces the efforts required during maintenance.

4.3 Annotations

Using Annotations, it is possible to configure and generate Resource data automatically without having to write methods for each getter and setter. The resource support reads through the properties set in the parameters of the variables and assembles a resource. The following are the annotations that were created:

a) @CJSON
This annotation is used to provide the resource's properties i.e. the name, prompt and access level to be used in the JSON response. The access level is an enumerator type with the following values AUTO, READ_ONLY, WRITE_ONLY, READ_WRITE and is used to control the way the parameters are accessed and added to the JSON response template.

```
@CJSON(name = "id", prompt = "ID", access = Access.READ_ONLY)
public Long id;
```

b) @CJSONRel
The annotation is used to define the parameters for a related resource. The resource support automatically generates a reference link that redirects the user to the resource.

```
@CJSONRel(name = "Conference",prompt="Conference",rel="Conference", access =
Access.AUTO)
public ConferenceResource conference;
```

c) @CJCommand
The properties of a CJCommand include a name, a prompt, the method, a precondition, and the data. The prompt and name are used to identify the command.

```
@CJCommand( precondition="APPROVED",name = "Update", prompt = "Update PO",rel =
"requestExtension",data = { @CJSON(access = Access.AUTO, name = "startDate", prompt
= "Start Date") , @CJSON(access = Access.AUTO, name = "endDate", prompt = "End
Date") }, method="put")
```

d) @CJQuery

This annotation is used to define Queries in Collection JSON. Queries are used for fetching specific information. E.g. one can filter purchase orders by EndDate.

```
@CJQuery(rel="filterByEndDate", prompt="Filter by endDate", data={@CJSON(name =
"endDate", prompt = "End Date", access = Access.AUTO )})
```

These annotations reduce the possibility of having errors because they qualify a configuration, which can be set for as many different options as possible as seen in Appendix VI. In the event that one wishes to extend the annotations to add extra features, it is possible to extend and implement additional solutions.

4.4 The Client

The client should be able to render new links easily and dynamically. Collection +JSON is able to do this by reading the API elements, and interpreting them into the required controls. As seen in Figure below, the client is able to interpret the links, commands, items and templates by parsing the definitions made in the API as seen in figure 19. Below is an example of a list of Conferences. The template provided makes it possible to create a new conference with the use of the Add form. Collection +JSON already provides the CRUD functionality.

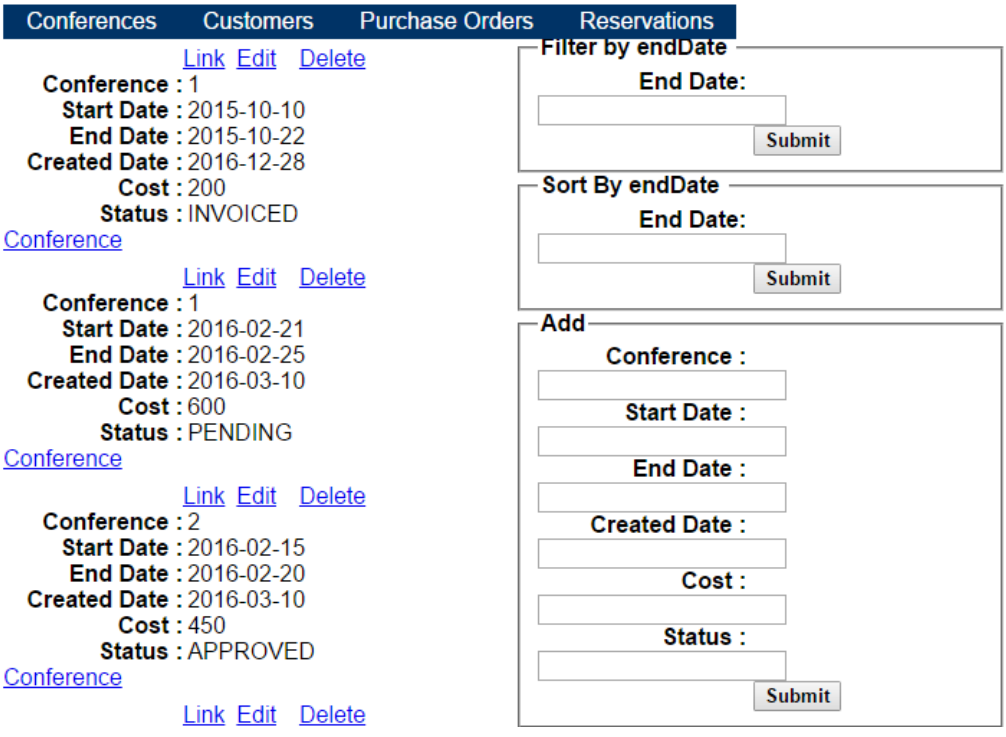
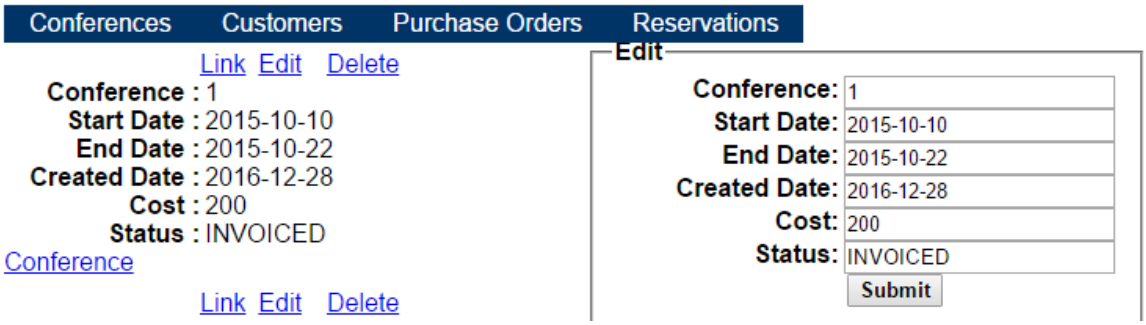
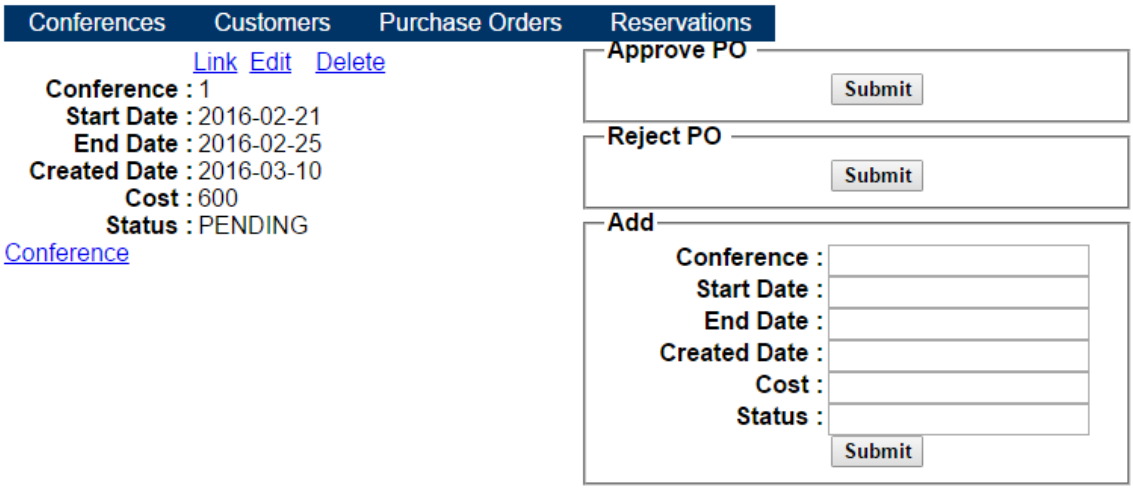


Figure 19 viewing a list of Conferences

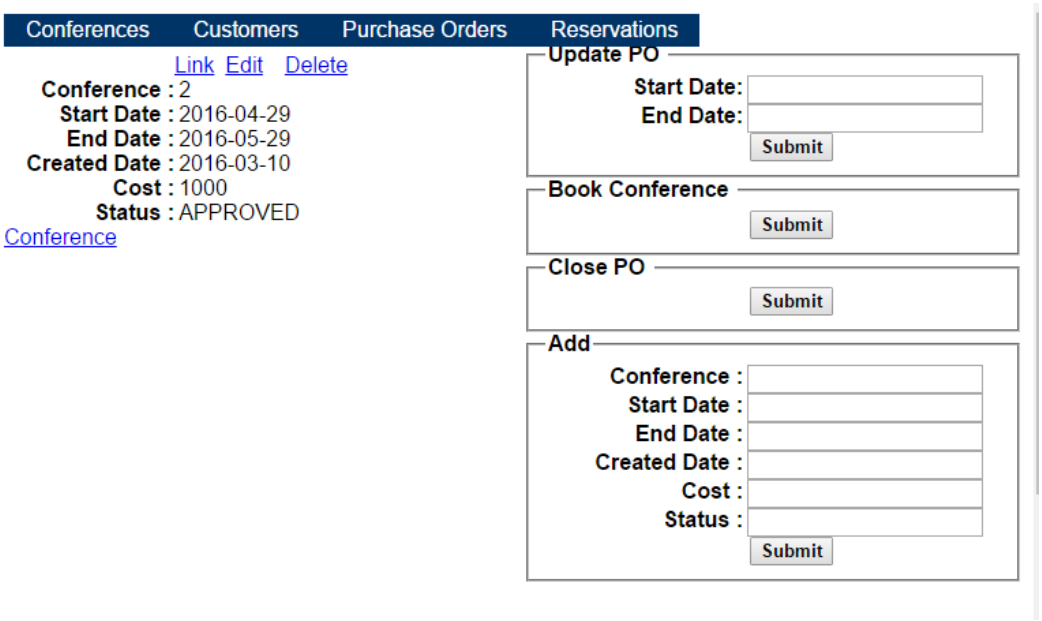
It is also possible to edit the data of the conference as seen in figure 19 by clicking on the edit link, which opens up a form that makes it possible to edit individual conferences. We can then update the conference information and click submit which updates the conference accordingly.



The client is able to expose the commands and render them as individual links. As Seen below, we are presented with the options of approving or rejecting a purchase order.



Some commands come with additional update information. The example below shows what happens when requesting for the duration of the booking by updating the start Date or end Date of the conference.



The Client is able to render the required fields to collect the update information. For example for the Update PO request, the client generates a start Date and end Date field.

5 DISCUSSION

In this section, we will discuss the overall results of the applied contribution and we will also look into the advantages, limitations and future work that could not be implemented within the scope of this project.

5.1 Advantages

In summary, the scope of our contribution was focused on improving and exposing richer update operations via REST update operations i.e. through PUT and POST, in the Collection +JSON hypermedia format, as opposed to duplicating the same update operations in the presentation layer.

a) **Generic Controller and true CRUD API**

As a result, we were able to implement a prototype with a generic controller that implements all Http methods in a way that could be inherited by other controllers. We were also able to implement commands that would be used to expose the actions that could be executed against a resource to change its state without breaking the simplicity of the REST CRUD properties. As seen below, the previous HAL implementation would require that a user implements methods for each of the URLs listed.

HTTP Method	URL	Payload	Result
POST	/api/orders	CreateNewOrder	Creates New Order Item
GET	/api/orders/{id}	GetOrder	Returns Order Item
GET	/api/orders	GetOrders	Get All Orders
PUT	/api/orders/	UpdateOrder	UpdateOrders
PUT	/api/orders/updateSeat	UpdateSeat	Update order seat information.
PUT	/api/orders/requestPOUpdate	UpdateDuration	Change start or end dates.
DELETE	/api/orders/{id}	DeleteOrder	DeleteOrders

We are able to solve the problem of using multiple arbitrary verbs and have a true CRUD REST API that is able to implement multiple methods that only uses single verbs.

The prototype shows that the solution is feasible and that it exposes the domains commands effectively.

HTTP Method	URL	Payload	Supported methods	Result
POST	/api/orders	createEntity	- createEntity	Creates New Order Item
GET	/api/orders/{id}	getEntity	-getAllEntities	Get single Orders
PUT	/api/orders/{id}	updateEntity	- requestPOUpdate, - acceptPo, - rejectPo,	UpdateOrders
DELETE	/api/orders/{id}	deleteEntity		DeleteOrders

b) Improved productivity with less code

It is evident from the implementation that there is less code required to implement controllers or implement hypermedia controls in the client because they are automatically generated in the API. The generic controller alleviates the requirement of always implementing the same methods in child controllers. One of the open issues left for future work is to present a metric for the productivity achieved after implementing this approach.

5.2 Limitations

In this section we discuss the various issues and limitations that the current solution has that have not been solved in the scope of this research.

a) Annotations may cause clutter

It is also worth noting that, annotations which were once the desired solution for making configuration of the resource easier, also introduce clutter into the applications resource file as seen in Appendix VI. For resources that may have multiple command operations with a lot more complex logic and a larger resource state life cycle, there may be a lot of cluttered annotations especially for commands that have multiple update parameters as seen below.

```
@CJCommand( precondition="APPROVED",name = "Update", prompt = "Update PO",rel =
"requestExtension",data = { @CJSON(access = Access.AUTO, name = "startDate", prompt
= "Start Date") , @CJSON(access = Access.AUTO, name = "endDate", prompt = "End
Date") }, method="put")
```

5.3 Future Work

The current solution is not perfect, and requires additional research and development to drastically improve on various aspects such as performance, legibility, productivity, security among others. In this section we discuss the works open for further development in the future.

First and foremost, there still remains room for improving Read methods which are used to create the criteria's used to filter out specific resources. The filtering implementation would make it easier for users to search for information and also define the keywords to be used for search and filter purposes.

Currently, the Contoso solution was implemented using the spring framework IDE that is based on Eclipse. The project consists of reusable methods that can be packaged into a library. This would be very useful in creating other projects and can be redistributed as a solution when applying the techniques discussed in this research.

The solution also currently has a client with a very basic interface that displays the collections as lists with raw hyperlinks and very simple forms with minimalistic characteristics. The client in Collection +JSON is able to read and dynamically generate the most suitable elements and controls to manipulate the data for CRUD purposes. There is room for improving this functionality by using Angular JS, html 5 and JavaScript to implement more robust interfaces. It is also possible to implement more advanced controls such as check boxes, drop down menus, search boxes among other controls which are more advanced and user friendly for capturing and displaying content in a neater manner. We will also implement well-designed tables that recognize the collections and create different columns that reflect the templates provided for each item in the collection.

The current solution supports partial Exception handling. Collection +JSON implements errors as elements, which are generated in the event that an exception is caught. Error messages contain a tittle, message and code. The solution currently implements basic exception handling

which entails having a custom exception. However, the solution does not generate error responses when methods fail to fully process information in the business logic. This can be further improved to handle all exceptions with little configuration.

Also, in the current approach, API security has not been implemented. Security of data is a wider scope that is much needed when exposing information. Having the ability to describe the layers of security and access control for different types of users should be done with ease, with little configuration. This has been left for future development to structure the user's access to data.

6 CONCLUSION

The main objective of this thesis was to reconcile the simplicity of REST with the rich concepts of DDD and CQRS. From our discussion, we were able to conclude that in order to achieve a robust API, it is important to have to make the right selection of a JSON Hypermedia format that promotes the constraints of REST. We concluded that Collection +JSON is the most suitable format because it provides a variety of rich and well-structured features such as templates, queries, links, commands and it also represents single items as well. It also does a good job when representing data collections. We were able to extend the Collection +JSON hypermedia type to map out the methods in the domain as commands. We also concluded that having RPC-styled resources with a URL like ***/api/conference/{id}/rename*** is not suitable because it goes against REST's resource-oriented presentation even though it seems to remove the need for the arbitrary verbs.

We discussed how to make use of DDD techniques to further improve the process of development by minimizing the redundant code found in controller methods. We also borrowed concepts from CQRS and used them to segregate the methods in the domain service layer into commands and queries. It is through this process that we are able to implement the right interface that allows accessing the update methods via a single PUT or POST Http request as opposed to using custom verbs that require a Http method for every request.

We were able to create a proof of concept application that implements Collection +JSON using Java on a spring boot MVC framework. Spring framework [15] is a Java web application framework that is used to create web applications with an intelligent library management that is able to configure new libraries easily.

[15] https://projects.spring.io/spring-framework/

7 REFERENCES

[1] R. T. Fielding, Architectural styles and the design of network-based software architectures, University of California,: Irvine, 2000.

[2] S. B. Balaji Varanasi, Spring REST, Rest and Webservices development with Spring, Salt Lake City: Appress, 2015.

[3] R. L. S. R. Amundsen M, RESTful Web APIs, O'Reilly Media Inc., 2015.

[4] L. R. a. M. Amundsen, Restful Web APIs, O'Reilly Media inc., 2013.

[5] K. Sookocheff, "On choosing a hypermedia type for your API - HAL, JSON-LD, Collection+JSON, SIREN, Oh My!," Red Finch Software, 10 6 2016. [Online]. Available: http://sookocheff.com/post/api/on-choosing-a-hypermedia-format/. [Accessed 10 06 2016].

[6] K. Sookocheff, "How REST Constraints Affect API Design," 19 03 2014. [Online]. Available: https://sookocheff.com/post/api/how-rest-constraints-affect-api-design/. [Accessed 13 03 2017].

[7] A. Kheyrollahi, "Exposing CQRS Through a RESTful API," 9 12 2013. [Online]. Available: https://www.infoq.com/articles/rest-api-on-cqrs. [Accessed 01 07 2016].

[8] D. Bets, J. Dominguez, G. Melnik, F. Simonazzi and M. Subramanian, "Exploring CQRS and Event Sourcing," in *A journey into high scalability, availability,and maintainability with Windows Azure*, Boston, Microsoft, 2012, pp. 212-240.

[9] R. Fielding, "REST APIs must be hypertext-driven," http://roy.gbiv.com, 20 october 2010. [Online]. Available: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. [Accessed 1 May 2017].

[10] M. Bieg, "A Web-based Tool to Semi-automatically Import Data from Generic REST APIs," 1 November 2014. [Online]. Available: http://mb21.github.io/api-explorer/a-web-based-tool-to-semi-automatically-import-data-from-generic-rest-apis.html. [Accessed 16 May 2017].

[11] M. Kelly, "HAL - Hypertext Application Language," 18 09 2013. [Online]. Available: http://stateless.co/hal_specification.html. [Accessed 13 04 2017].

[12] M. Amundsen, in *Building Hypermedia APIs with HTML5 and Node*, O'Reilly Media, Inc., 2011, pp. 15-25.

[13] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Boston: Addison Wesley, 2003.

[14] S. Penchikala, "Domain Driven Design and Development In Practice," InfoQ, 12 June 2008. [Online]. Available: https://www.infoq.com/articles/ddd-in-practice. [Accessed 04 January 2017].

[15] D. . Esposito and A. Saltarello, "Introducing CQRS," 09 October 2014. [Online]. Available: https://www.microsoftpressstore.com/articles/article.aspx?p=2248809. [Accessed 17 May 2017].

[16] R. Laddad, “AOP and metadata: A perfect match, Part 1,” 19 December 2016. [Online]. Available: https://www.ibm.com/developerworks/java/library/j-aopwork3/j-aopwork3-pdf.pdf. [Accessed 15 May 2017].

[17] B. Meyer, Object Oriented Software construction, Santa Barbara(Carlifornia): ISE Inc, 1997.

[18] I. Nadareishvili, “All Content Management Systems (CMS) Are Broken In a Bad Way.,” Irakli Nadareishvili, 22 October 2013. [Online]. Available: http://www.freshblurbs.com/blog/2013/10/22/web-of-apis-hypermedia-collection-document.html. [Accessed 2 May 2017].

[19] V. Sahni, “Best Practices for Designing a Pragmatic RESTful API,” 2015. [Online]. Available: http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api#requirements. [Accessed 20 10 2016].

[20] “DOMAIN-DRIVEN DESIGN EXAMPLE,” 22 4 2013. [Online]. Available: https://www.mirkosertic.de/blog/2013/04/domain-driven-design-example/. [Accessed 10 10 2016].

[21] K. Sookocheff, “How REST Constraints Affect API Design,” 19 03 2015. [Online]. Available: https://sookocheff.com/post/api/how-rest-constraints-affect-api-design/. [Accessed 21 02 2017].

[22] https://opencredo.com/designing-rest-api-fine-grained-resources-hateoas-hal/, “Designing a REST API with fine-grained resources, HATEOAS and HAL,” 12 August 2015. [Online]. Available: https://opencredo.com/designing-rest-api-fine-grained-resources-hateoas-hal/. [Accessed 14 April 2017].

8 APPENDIX

I. SOURCE CODE

The prototype code for the implemented tool along with a Spring Java project where the generator is applied is submitted along with this thesis in an archived file. The source code is also maintained in a public bit-bucket repository which could be accessed at ***https://bitbucket.org/MikeMichuki/collection-JSON***.

II. BASIC COLLECTION +JSON RESPONSE

The response of a Basic Collection + JSON resource representation

```
GET http://contoso.com/orders

{
  "collection": {
    "version": "1.0",
    "href": "http://contoso.com/orders",
    "links": [ {
        "href": "http://contoso.com/customers",
        "rel": "Customers",
        "prompt": "Customers",
        "name": "Customers"
      }, {
        "href": "http://contoso.com/conferences",
        "rel": "conferences",
        "prompt": "Conferences",
        "name": "Conferences"
      } ],
    "items": [
      {
        "href": "http://contoso.com/orders",
        "data": [ {
            "name": "startDate",
            "prompt": "Start Date",
            "value": "2016-04-29"
          }, {
            "name": "status",
            "prompt": "Status",
            "value": "CONFERENCEDISPATCHED"
          } ],
        "links": [
          {
            "href": "http://contoso.com/conferences/3",
            "rel": "Conference",
            "prompt": "Conference",
            "name": "Conference"
          } ]   }
    ],
    "queries": [
      {
        "href": " http://contoso.com/pos",
        "rel": "filterByEndDate",
        "prompt": "Filter by endDate",
        "data": [
          {
            "name": "endDate",
            "prompt": "End Date",
            "value": ""
          }
        ]
      }
    "template": {
      "data": [ {
          "name": "startDate",
          "prompt": "Start Date",
          "value": ""
        }, {
          "name": "status",
          "prompt": "Status",
          "value": ""
        } ]
      }
   }
}
```

III. ADVANCED COLLECTION +JSON RESPONSE

The response of the improved Collection +JSON with Commands and their individual templates.


```
{
  "collection": {
    "version": "1.0",
    "href": "http://localhost:8686/api/pos",
    "links": [
      {
        "href": "http://localhost:8686/api/conferences",
        "rel": "conferences",
        "prompt": "Conferences",
        "name": "Conferences"
      },
      {
        "href": "http://localhost:8686/api/pos",
        "rel": "POs",
        "prompt": "Purchase Orders",
        "name": "PO"
      }
    ],
    "items": [
      {
        "href": "http://localhost:8686/api/pos/2",
        "data": [
          {
            "name": "id",
            "prompt": "ID",
            "value": "2"
          },
          {
            "name": "Conference",
            "prompt": "Conference",
            "value": "1"
          },
          {
            "name": "startDate",
            "prompt": "Start Date",
            "value": "2016-03-15"
          },
          {
            "name": "endDate",
            "prompt": "End Date",
            "value": "2016-03-23"
          },
          {
            "name": "cost",
            "prompt": "Cost",
            "value": "131234.0"
          },
          {
            "name": "status",
            "prompt": "Status",
            "value": "APPROVED"
          }
        ],
        "links": [
          {
            "href": "http://localhost:8686/api/conferences/1",
            "rel": "Conference",
            "prompt": "Conference",
            "name": "Conference"
          }
        ]
      }
    ],
```

```
    "commands": [
      {
        "href": "http://localhost:8686/api/pos/2",
        "rel": "requestExtension",
        "prompt": "Update PO",
        "name": "Update",
        "method": "put",
        "data": [
          {
            "name": "startDate",
            "prompt": "Start Date",
            "value": ""
          },
          {
            "name": "endDate",
            "prompt": "End Date",
            "value": ""
          }
        ]
      },
      {
        "href": "http://localhost:8686/api/pos/2",
        "rel": "dispatchConference",
        "prompt": "Book Conference",
        "name": "Dispatch",
        "method": "put"
      },
      {
        "href": "http://localhost:8686/api/pos/2",
        "rel": "closePurchaseOrder",
        "prompt": "Close PO",
        "name": "Close",
        "method": "put"
      }
    ],
    "template": {
      "data": [
        {
          "name": "Conference",
          "prompt": "Conference",
          "value": ""
        },
        {
          "name": "startDate",
          "prompt": "Start Date",
          "value": ""
        },
        {
          "name": "endDate",
          "prompt": "End Date",
          "value": ""
        },
        {
          "name": "cost",
          "prompt": "Cost",
          "value": ""
        },
        {
          "name": "status",
          "prompt": "Status",
          "value": ""
        }
      ]
    }
  }
}
```

IV. BASIC HAL CONTROLLER

Here is a typical controller that contains multiple methods.

```
@RestController
@RequestMapping("/rest/orders")
public class PurchaseOrderController {

        @Autowired
        PurchaseOrderManager poManager;

        @Autowired
        PurchaseOrderResourceAssembler poResourceAssembler;

        @RequestMapping("/{id}")
        public ResponseEntity<PurchaseOrderResource> getPO(@PathVariable("id")
        Long id) throws PONotFoundException {
                PurchaseOrder po = poManager.getPurchaseOrder(id);
                ...
        }

        @RequestMapping(method = RequestMethod.POST, headers =
        "Accept=application/JSON")
        @ResponseStatus(HttpStatus.CREATED)
        public ResponseEntity<PurchaseOrderResource> createPO(@RequestBody
        PurchaseOrderResource por)throws ConferenceNotAvailableException {
                ...
        }

        @RequestMapping(value = "/{id}/updateseatposition ", method =
        RequestMethod.PUT)
        public ResponseEntity<PurchaseOrderResource>
        updateSeatPosition(@PathVariable("id")
        Long id, @RequestBody PurchaseOrderResource por) {
                PurchaseOrder po = poManager.updateSeatPosition(id, por);
                ResponseEntity<PurchaseOrderResource> result = new
                ResponseEntity<PurchaseOrderResource>(
                                poResourceAssembler.toResource(po),
                HttpStatus.OK);
                return result;
        }

        @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
        public ResponseEntity<PurchaseOrderResource> updatePO(@PathVariable("id")
        Long id, @RequestBody PurchaseOrderResource por)
                        throws Exception {
                ...
        }
}
```

V. GENERIC CONTROLLER

The Generic Abstract controller

The complete Generic controller can be accessed in the source code found in Appendix A.

```
package util;

import util.interfaces.BaseService;
public abstract class GenericController<T, D> {
	public final BaseService<T, D> service;

	public GenericController(BaseService<T, D> service) {
		this.service = service;
	}

	@RequestMapping(method = RequestMethod.POST)
	public D createEntity(@RequestBody D collection) throws Exception {
		System.out.println(collection);
		service.createEntity(collection);
		return getAllEntities();
	}

	@RequestMapping(method = RequestMethod.DELETE, value = "/{id}")
	public D deleteEntity(@PathVariable("id") T id) throws Exception {
		service.deleteEntity(id);
		return getAllEntities();
	}

	@RequestMapping(method = GET, path = "")
	public D getAllEntities() throws Exception {
		return service.getAllEntities();
	}

	@RequestMapping(method = GET, path = "/{id}")
	public D getOneEntity(@PathVariable T id) throws Exception {
		return service.getSingleEntity(id);
	}

	@RequestMapping(method = RequestMethod.PUT,
	value = { "/{id}/*", "*","/{id}" })
	public D updateEntity(@PathVariable T id, @RequestBody D collection,
		@RequestHeader("domain-model") String command) throws Exception {
		if(command.equals("edit")){
			service.updateEntity(id, collection);}
		for (Method method : service.getClass().getDeclaredMethods()){
			if (method.getName().equals(command)) {
				Class<?>[] params = method.getParameterTypes();
				if (params.length == 1) {
					method.invoke(service,id);
				} else if (params.length == 0) {
					method.invoke(service);
				} else if(params.length > 1){
				System.out.print("JSON " + collection
				+ "Params " + params[0]);
				method.invoke(service,id,collection.toString());
				} else {
					throw new Exception("Command not found");
				}
			}
		}
		return getOneEntity(id);
	}
}
```

VI. PURCHASE ORDER RESOURCE

Below are the settings made in the purchase order resource file.

```
package com.app.resource;

@Getter
@Setter
@CJQuery(rel="filterByEndDate", prompt="Filter by endDate", data={@CJSON(name =
"endDate", prompt = "End Date", access = Access.AUTO )})
@CJQuery(rel="sortByEndDate", prompt="Sort By endDate", data={@CJSON(name =
"endDate", prompt = "End Date", access = Access.AUTO )})
@CJCommand( precondition="PENDING", name = "Approve", prompt = "Approve PO",rel =
"acceptPurchaseOrder",data = {}, method="put")
@CJCommand( precondition="PENDING",name = "Reject", prompt = "Reject PO",rel =
"rejectPurchaseOrder",data = {}, method="put")
@CJCommand( precondition="APPROVED",name = "Update", prompt = "Update PO",rel =
"requestExtension",data = { @CJSON(access = Access.AUTO, name = "startDate", prompt
= "Start Date") , @CJSON(access = Access.AUTO, name = "endDate", prompt = "End
Date") }, method="put")
@CJCommand( precondition="REJECTED",name = "Update", prompt = "Update PO",rel =
"updatePurchaseOrder",data = {}, method="put")
@CJCommand( precondition="APPROVED",name = "Dispatch", prompt = "Book
Conference",rel = "dispatchConference",data = {}, method="put")
@CJCommand( precondition="DISPATCHED",name = "Reject", prompt = "Reject
Conference",rel = "rejectConference",data = {}, method="put")
@CJCommand( precondition="DISPATCHED",name = "Deliver", prompt = "Confirm
Conference",rel = "deliverConference",data = {}, method="put")
@CJCommand( precondition={"DELIVERED","REJECTED"},name = "Return", prompt = "Return
Conference",rel = "returnConference",data = {}, method="put" )
@CJCommand( precondition="APPROVED",name = "Close", prompt = "Close PO",rel =
"closePurchaseOrder",data = {}, method="put")
@CJCommand( precondition="RETURNED",name = "Invoice", prompt = "Invoice PO",rel =
"confirmInvoiceSent",data = {}, method = "put")
public class PurchaseOrderResource extends ResourceSupport<Long> {
        public PurchaseOrderResource() {
                super(PurchaseOrderController.class);
        }

        @CJSON(name = "id", prompt = "ID", access = Access.READ_ONLY)
        public Long id;

        @CJSONRel(name = "Conference",prompt="Conference",rel="Conference", access
= Access.AUTO)
        public ConferenceResource conference;

        @CJSON(name = "startDate", prompt = "Start Date", access = Access.AUTO)
        public LocalDate startDate;

        @CJSON(name = "endDate", prompt = "End Date",access = Access.AUTO)
        public LocalDate endDate;

        @CJSON(name = "createdDate", prompt = "Created Date",access = Access.AUTO)
        public LocalDate createdDate;

        @CJSON(name = "cost", prompt = "Cost", access = Access.AUTO)
        public BigDecimal cost;

        @CJSON(name = "status", prompt = "Status", access = Access.READ_WRITE)
        public Status status;

        public Long getId() {
                return id;
        }
}
```

VII. SIREN RESPONSE

Below is an example of a resource that is returned by a Siren API.

```
GET http://contoso.com/orders/1234

{
    "class": "order",
    "links": [
        {"rel": [ "self" ], "href": "http://contoso.com/orders/1234 "},
        {"rel": [ "next" ], "href": "http://contoso.com/orders/1234?page=2"}
    ],
    "actions": [{
        "class": "add-order",
        "href": "http://contoso.com/orders/1234",
        "method": "POST",
        "fields": [
            {"name": "name", "type": "string"},
            {"name": "alternateName", "type": "string"},
            {"name": "image", "type": "href"}        ]
    }],
    "properties": {
        "size": "2"
    },
    "entities": [
        {
            "links": [
                {"rel": [ "self" ], "href": "http://contoso.com/orders/123012"},
                {"rel": [ "orders" ], "href":
                 "http://contoso.com/orders/12435/orders"}
            ],
            "properties": {
                 "idres": "4",
                 "name": "Alex wong",
                 "price": "490"
            }
        },
        { "links": [
                {"rel": [ "self" ], "href": "http://contoso.com/orders/1234"},
                {"rel": [ "purchaseorders" ], "href":
                 "http://contoso.com/orders/12334" }
            ],"properties": {
                 "idres": "4",
                 "name": "Alex wong",
                 "price": "490"           }
        }
    ]
}
```

VIII. LICENSE

Non-exclusive licence to reproduce thesis and make thesis public.

I, **Michael Ngugi Michuki**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Exposing Rich Update Operations via REST APIs,

supervised by Luciano Garcia Bañuelos ,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **23.05.2017**