Build a REST Web Service Using JAX-RS and MongoDB

This Lab provides a step-by-step guide for building and configuring a REST web service in Java JAX-RS, Jersey and MongoDB.

Prerequisites:

- Spring Tool Suite 3/4
- Java 1.8
- Tomcat 8.0
- MongoDB Compass Community
- Postman

The Spring Tool Suite is an open-source, Eclipse-based IDE distribution that provides a superset of the Java EE distribution of Eclipse

Business Scenario:

An application has to be developed for management of interns hired by Yash technologies. The application will provide following services:

1. Based on technical evaluation, intern will be hired and details will be entered in system. Interns technical level will be decided based on college semester marks. Interns will have to go through training based on levels and clear certifications.
2. Intern's details can be retrieved by HR to assign projects to intern.
3. Intern's personal details can be updated.
4. Intern's level can be updated based on semester scores.
5. Intern's details will be removed from system if intern has completed internship.

There will be two applications to suffice need of above system.

InternsUIApp will be developed using Angular/ReactJS and InternsBusinessApp will be business application.

Web Service

Web service is a system that enables applications to communicate with an API. Web service helps to expose business logic through an API interface where different systems communicate over network. At higher level there are two parties involved, party providing the service is web service provider and the one utilizing it is web service consumer.

REST Web Service

REST stands for Representational State Transfer. REST was a term coined by Roy Fielding in his doctoral dissertation. It is an architecture style for for creating network based applications. Key properties of REST are client-server communication, stateless protocol, cacheable, layered implementation and uniform interface. In many ways, the World Wide Web itself, which is based on HTTP, is the best example of REST-based architecture. These days, REST is used everywhere - from desktops to mobiles and even in the likes of Facebook, Google, and Amazon.

There are six guiding constraints of REST. These are:

Client-Server Mandate
This mandate underscores the fact that REST is a distributed approach via the nature of separation between client and server. Each service has multiple capabilities and listens for requests. Requests are made by a consumer and accepted or rejected by the server.

Statelessness
Due to the nature of statelessness, it is a guiding principle of RESTful architecture. It mandates what kind of commands can be offered between client and server. Implementing stateless requests means the communication between consumer and service is initiated by the request, and the request contains all the information necessary for the server to respond.

Cache
Cache mandates that server responses be labelled as either cacheable or not. Caching helps to mitigate some of the constraints of statelessness. For example, a request that is cached by the consumer in an attempt to avoid re-submitting the same request twice.

Interface / Uniform Contract
RESTful architecture follows the principles that define a Uniform Contract. This prohibits the use of multiple, self-contained interfaces within an API. Instead, one interface is distributed by hypermedia connections.

Layered System
This principle is the one that makes RESTful architecture so scalable. In a Layered System, multiple layers are used to grow and expand the interface. None of the layers can see into the other. This allows for new commands and middleware to be added without impacting the original commands and functioning between client and server.

Optional: Code-On-Demand
RESTful applications don't have to include Code-On-Demand, but they must have Client-Server, Statelessness, Caching, Uniform Contract, and Layered Systems. Code-on-Demand allows logic within clients to be separate from that within servers. This allows them to be updated independently of server logic.

REST refers to a set of defining principles for developing API. It uses HTTP protocols like GET, PUT, POST to link resources to actions within a client-server relationship. In addition

to the client-server mandate, it has several other defining constraints. The principles of RESTful architecture serve to create a stable and reliable application that offers simplicity and end-user satisfaction.

CRUD: Foundation and Principles

CRUD is an acronym for CREATE, READ, UPDATE, DELETE. These form the standard database commands that are the foundation of CRUD.
CRUD's origins are in database records.
By definition, CRUD is more of a cycle than an architectural system. On any dynamic website, there are likely multiple CRUD cycles that exist.
For instance, a buyer on an eCommerce site can CREATE an account, UPDATE account information, and DELETE things from a shopping cart.
A Warehouse Operations Manager using the same site can CREATE shipping records, RETRIEVE them as needed, and UPDATE supply lists. Retrieve is sometimes substituted for READ in the CRUD cycle.

Database Origins

The CRUD cycle is designed as a method of functions for enhancing persistent storage—with a database of records, for instance. As the name suggests, persistent storage outlives the processes that created it. These functions embody all the hallmarks of a relational database application.

In modern software development, CRUD has transcended its origins as foundational functions of a database and now maps itself to design principles for dynamic applications like HTTP protocol, DDS, and SQL.

Principles of CRUD

As mentioned above, the principles of the CRUD cycle are defined as CREATE, READ/RETRIEVE, UPDATE, and DELETE.

Create
CREATE procedures generate new records via INSERT statements.

Read/Retrieve
READ procedures reads the data based on input parameters. Similarly, RETRIEVE procedures grab records based on input parameters.

Update
UPDATE procedures modify records without overwriting them.

Delete
DELETE procedures delete where specified.

REST and CRUD Similarities

If you look at the two as we have described above, it may be difficult to understand why they are often treated in the same way. REST is a robust API architecture and CRUD is a cycle for keeping records current and permanent. The lack of clarity between the two is lost for many when they fail to determine when CRUD ends and REST begins. We mentioned above that CRUD can be mapped to DDS, SQL, and HTTP protocols. And that HTTP protocols are the link between resources in RESTful architecture, a core piece of REST's foundation.

Mapping CRUD principles to REST means understanding that GET, PUT, POST and CREATE, READ, UPDATE, DELETE have striking similarities because the former grouping applies the principles of the latter. However, it is also important to note that a RESTful piece of software architecture means more than mapping GET, PUT, POST commands.

REST and CRUD: What's the Difference?

CRUD is a cycle that can be mapped to REST, by design. Permanence, as defined in the context of CRUD, is a smart way for applications to mitigate operational commands between clients and services. But REST governs much more than permanence within its principles of architecture.

Here are some of the ways that REST is not only different than CRUD but also offers much more:

- REST is an architectural system centered around resources and hypermedia, via HTTP protocols
- CRUD is a cycle meant for maintaining permanent records in a database setting
- CRUD principles are mapped to REST commands to comply with the goals of RESTful architecture

In REST:

- Representations must be uniform with regard to resources
- Hypermedia represents relationships between resources
- Only one entry into an API to create one self-contained interface, then hyperlink to create relationships

Developers select REST for a number of reasons:

1. Performance
2. Scalability
3. Simplicity
4. Modifiability
5. Portability

6. Reliability

7. Visibility

REST is sometimes seen as an alternate for SOAP.

REST vs SOAP

Roy Fielding was a member of the team that wrote the specification for HTTP and co-founder of Apache HTTP server project. He introduced the word REST and the concept in his doctoral thesis in 2000. REST disruptively took over as an architectural style for web services implementation.

- SOAP stands for Simple Object Access Protocol. REST stands for REpresentational State Transfer.
- SOAP is a XML based messaging protocol and REST is not a protocol but an architectural style.
- SOAP has a standard specification but there is none for REST.
- Whole of the web works based on REST style architecture. Consider a shared resource repository and consumers access the resources.
- Even SOAP based web services can be implemented in RESTful style. REST is a concept that does not tie with any protocols.
- SOAP is distributed computing style and REST is web style (web is also a distributed computing model).
- REST messages should be self-contained and should help consumer in controlling the interaction between provider and consumer(example, links in message to decide the next course of action). But SOAP doesn't has any such requirements.

- REST does not enforces message format as XML or JSON or etc. But SOAP is XML based message protocol.
- REST follows stateless model. SOAP has specifications for stateful implementation as well.
- SOAP is strongly typed, has strict specification for every part of implementation. But REST gives the concept and less restrictive about the implementation.
- Therefore REST based implementation is simple compared to SOAP and consumer understanding.

- SOAP uses interfaces and named operations to expose business logic. REST uses (generally) URI and methods like (GET, PUT, POST, DELETE) to expose resources.
- SOAP has a set of standard specifications. WS-Security is the specification for security in the implementation. It is a detailed standard providing rules for security in application implementation. Like this we have separate specifications for messaging, transactions, etc. Unlike SOAP, REST does not has dedicated concepts for each of these. REST predominantly relies on HTTPS.
- Above all both SOAP and REST depends on design and implementation of the application.

Features of a RESTful Services

Every system uses resources. These resources can be pictures, video files, Web pages, business information, or anything that can be represented in a computer-based system. The purpose of a service is to provide a window to its clients so that they can access these resources. Service architects and developers want this service to be easy to implement, maintainable, extensible, and scalable. A RESTful design promises that and more.

In general, RESTful services should have following properties and features:

- Representations
- Messages
- URIs
- Uniform interface
- Stateless
- Links between resources
- Caching

Representations

The focus of a RESTful service is on resources and how to provide access to these resources. A resource can easily be thought of as an object as in OOP. A resource can consist of other resources. While designing a system, the first thing to do is identify the resources and determine how they are related to each other. This is similar to the first step of designing a database: Identify entities and relations.

Once we have identified our resources, the next thing we need is to find a way to represent these resources in our system. You can use any format for representing the resources, as REST does not put a restriction on the format of a representation.

For example, depending on your requirement, you can decide to use JSON or XML. If you are building Web services that will be used by Web pages for AJAX calls, then JSON is a good choice. XML can be used to represent more complex resources. For example a resource called "Interns" can be represented as:

JSON representation of a resource

```
{
    "id": "1001",
```

```
    "internFirstName": "Rob",
    "internLastName": "Johnson”,
    “internAge”:21,
    "level": "BEGINNER"
}
```

XML representation of a resource

```
<interns>
<id>1001</id>
<internFirstName>Rob</internFirstName>
<internLastName>Johnson</internLastName>
<internAge>21</internAge>
<level>BEGINNER</level>
</interns>
```

In fact, you can use more than one format and decide which one to use for a response depending on the type of client or some request parameters. Whichever format you use, a good representation should have some obvious qualities:

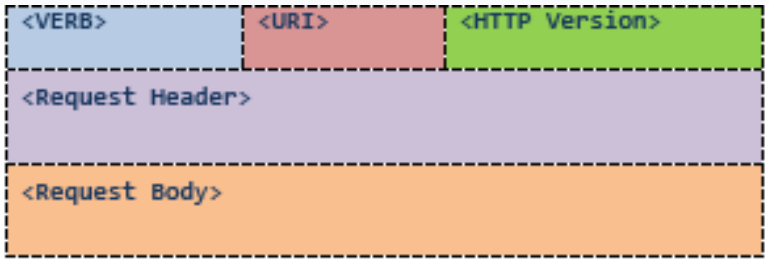
- Both client and server should be able to comprehend this format of representation.
- A representation should be able to completely represent a resource. If there is a need to partially represent a resource, then you should think about breaking this resource into child resources. Dividing big resources into smaller ones also allows you to transfer a smaller representation. Smaller representations mean less time required to create and transfer them, which means faster services.
- The representation should be capable of linking resources to each other. This can be done by placing the URI or unique ID of the related resource in a representation (more on this in the coming sections).

Messages

The client and service talk to each other via messages. Clients send a request to the server, and the server replies with a response. Apart from the actual data, these messages also contain some metadata about the message. It is important to have some background about the HTTP 1.1 request and response formats for designing RESTful Web services.

HTTP Request

An HTTP request has the format shown in Figure



HTTP request format.

<VERB> is one of the HTTP methods like GET, PUT, POST, DELETE, OPTIONS, etc

<URI> is the URI of the resource on which the operation is going to be performed

<HTTP Version> is the version of HTTP, generally "HTTP v1.1" .

<Request Header> contains the metadata as a collection of key-value pairs of headers and their values. These settings contain information about the message and its sender like client type, the formats client supports, format type of the message body, cache settings for the response, and a lot more information.

<Request Body> is the actual message content. In a RESTful service, that's where the representations of resources sit in a message.

There are no tags or markups to denote the beginning or end of a section in an HTML message.

Listing Three is a sample POST request message, which is supposed to insert a new resource Interns.

A sample POST request

```
POST http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns

Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 123
<?xml version="1.0" encoding="utf-8"?>
<interns>
<id>1001</id>
<internFirstName>Rob</internFirstName>
<internLastName>Johnson</internLastName>
<internAge>21</internAge>
<level>BEGINNER</level>
</interns>
```

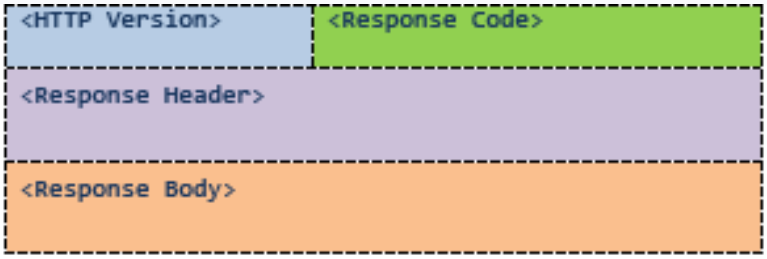
You can see the POST command, which is followed by the URI and the HTTP version. This request also contains some request headers. Host is the address of the server. Content-Type tells about the type of contents in the message body. Content-Length is the length of the data in message body. Content-Length can be used to verify that the entire message body has been received. Notice there are no start or end tags in this message.

A GET request.

```
GET http://www.w3.org/Protocols/rfc2616/rfc2616.html HTTP/1.1
Host: www.w3.org
Accept: text/html,application/xhtml+xml,application/xml; …
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 …
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,hi;q=0.6
```

There is no message body in this request. The Accept header tells the server about the various presentation formats this client supports. A server, if it supports more than one representation format, it can decide the representation format for a response at runtime depending on the value of the Accept header. User-Agent contains information about the type of client who made this request. Accept-Encoding/Language tells about the encoding and language this client supports.

HTTP Response



The server returns <response code>, which contains the status of the request. This response code is generally the 3-digit HTTP status code.

<Response Header> contains the metadata and settings about the response message.

<Response Body> contains the representation if the request was successful.

HTTP Method:

The PUT, GET, POST and DELETE methods are typical used in REST based architectures. The following table gives an explanation of these operations.

- GET defines a reading access of the resource without side-effects. The resource is never changed via a GET request, e.g., the request has no side effects (idempotent).
- PUT creates a new resource. It must also be idempotent.
- DELETE removes the resources. The operations are idempotent. They can get repeated without leading to different results.
- POST updates an existing resource or creates a new resource.

Resources are uniquely identified using URI. Web services that conforms to the constraints of REST are called RESTful web services.

Addressing Resources

REST requires each resource to have at least one URI. A RESTful service uses a directory hierarchy like human readable URIs to address its resources. The job of a URI is to identify a resource or a collection of resources. The actual operation is determined by an HTTP verb. The URI should not say anything about the operation or action. This enables us to call the same URI with different HTTP verbs to perform different operations.

Suppose we have a database of persons and we wish to expose it to the outer world through a service. A resource person can be addressed like this:

http://Interns-management/Interns/1001

This URL has following format: Protocol://ServiceName/ResourceType/ResourceID

Here are some important recommendations for well-structured URIs:

- Use plural nouns for naming your resources.
- Be concise
- Be easy to remember
- Avoid using spaces as they create confusion. Use an _ (underscore) or – (hyphen) instead.
- A URI is case insensitive. I use camel case in my URIs for better clarity. You can use all lower-case URIs.
- You can have your own conventions, but stay consistent throughout the service. Make sure your clients are aware of this convention. It becomes easier for your clients to construct the URIs programmatically if they are aware of the resource hierarchy and the URI convention you follow.
- A cool URI never changes; so give some thought before deciding on the URIs for your service. If you need to change the location of a resource, do not discard the old URI. If a request comes for the old URI, use status code 300 and redirect the client to the new location.
- Avoid verbs for your resource names until your resource is actually an operation or a process. Verbs are more suitable for the names of operations.

For example, a RESTful service should not have the URIs http://Interns-management/FetcthIntern/1 or http://interns-management/DeleteIntern?id=1001

Query Parameters in URI

The preceding URI is constructed with the help of a query parameter:

http://interns-management/interns?id=1001

The query parameter approach works just fine and REST does not stop you from using query parameters. However, this approach has a few disadvantages.

- Increased complexity and reduced readability, which will increase if you have more parameters
- Search-engine crawlers and indexers like Google ignore URIs with query parameters. If you are developing for the Web, this would be a great disadvantage as a portion of your Web service will be hidden from the search engines.

The basic purpose of query parameters is to provide parameters to an operation that needs the data items. For example, if you want the format of the presentation to be decided by the client. You can achieve that through a parameter like this:

http://interns-management/yash-interns/1001?format=xml&encoding=UTF8

or

http://interns-management/yash-interns/1001?format=json&encoding=UTF8

Including the parameters format and encoding here in the main URI in a parent-child hierarchy will not be logically correct as they have no such relation:

http://interns-management/yash-interns/1001/json/UTF8

Query parameters also allow optional parameters. This is not otherwise possible in a URI. You should use query parameters only for the use they are intended for: providing parameter values to a process.

As REST has become the default for most Web and mobile apps, it's imperative to have the basics at your fingertips.

Uniform Interface

RESTful systems should have a uniform interface. HTTP 1.1 provides a set of methods, called verbs, for this purpose. Among these the more important verbs are:

Method	Operation performed on server	Quality
GET	Read a resource.	Safe
PUT	Insert a new resource or update if the resource already exists.	Idempotent
POST	Insert a new resource. Also can be used to update an existing resource.	N/A
DELETE	Delete a resource .	Idempotent
OPTIONS	List the allowed operations on a resource.	Safe
HEAD	Return only the response headers and no response body.	Safe

A Safe operation is an operation that does not have any effect on the original value of the resource. For example, the mathematical operation "divide by 1" is a safe operation because no matter how many times you divide a number by 1, the original value will not change. An Idempotent operation is an operation that gives the same result no matter how many times you perform it. For example, the mathematical operation "multiply by zero" is idempotent because no matter how many times you multiply a number by zero, the result is always same. Similarly, a Safe HTTP method does not make any changes to the resource on the server. An Idempotent HTTP method has same effect no matter how many times it is performed. Classifying methods as Safe and Idempotent makes it easy to predict the results in the unreliable environment of the Web where the client may fire the same request again.

GET is probably the most popular method on the Web. It is used to fetch a resource.

HEAD returns only the response headers with an empty body. This method can be used in a scenario when you do not need the entire representation of the resource. For example, HEAD can be used to quickly check whether a resource exists on the server or not.

The method OPTIONS is used to get a list of allowed operations on the resource. For example consider the request:

```
1    OPTIONS http://interns-management/yash-interns/1001 HTTP/1.1
2    HOST: interns-management
```

The service after authorizing and authenticating the request can return something like:

```
1    200 OK
2    Allow: HEAD, GET, PUT
```

The second line contains the list of operations that are allowed for this client.

You should use these methods only for the purpose for which they are intended. For instance, never use GET to create or delete a resource on the server. If you do, it will confuse your clients

and they may end up performing unintended operations. To illustrate, this let's consider this request:

```
1    GET http://intern-management/DeleteIntern/1001 HTTP/1.1
2    HOST: intern-management
```

By HTTP 1.1 specification, a GET request is supposed to fetch resources from the server. But it is very easy to implement your service such that this request actually deletes a Intern. This request may work perfectly, but this is not a RESTful design. Instead, use the DELETE method to delete a resource like this:

```
1    DELETE http://intern-management/interns/1001 HTTP/1.1
2    HOST: intern-management
```

REST recommends a uniform interface and HTTP provides you that uniform interface. However, it is up to service architects and developers to keep it uniform.

Difference between PUT and POST

The short descriptions of these two methods I provided above are almost the same. These two methods confuse a lot of developers. So let's discuss these separately.

The key difference between PUT and POST is that PUT is idempotent while POST is not. No matter how many times you send a PUT request, the results will be same. POST is not an idempotent method. Making a POST multiple times may result in multiple resources getting created on the server.

Another difference is that, with PUT, you must always specify the complete URI of the resource. This implies that the client should be able to construct the URI of a resource even if it does not yet exist on the server. This is possible when it is the client's job to choose a unique name or ID for the resource, just like creating a user on the server requires the client to choose a user ID. If a client is not able to guess the complete URI of the resource, then you have no option but to use POST.

Request	Operation
`PUT http://intern-management/yash-interns/`	Won't work. `PUT` requires a complete URI
`PUT http://intern-management/yash-interns/1001`	Insert a new person with `InternId=1001` if it does not already exist, or else update the existing resource
`POST http://intern-management/yash-interns/`	Insert a new person every time this request is made and generate a new `InternId`.
`POST http://intern-management/yash-interns/1001`	Update the existing person where `InternId=1001`

It is clear from the above table that a PUT request will not modify or create more than one resource no matter how many times it is fired (if the URI is same). There is no difference between PUT and POST if the resource already exists, both update the existing resource. The third request (POST http://intern-management/yash-interns/) will create a resource each time it is

fired. A lot of developers think that REST does not allow POST to be used for update operation; however, REST imposes no such restrictions.

Statelessness

A RESTful service is stateless and does not maintain the application state for any client. A request cannot be dependent on a past request and a service treats each request independently. HTTP is a stateless protocol by design and you need to do something extra to implement a stateful service using HTTP. But it is really easy to implement stateful services with current technologies. We need a clear understanding of a stateless and stateful design so that we can avoid misinterpretation.

A stateless design looks like so:

Request1: `GET http://interns-management/yash-interns/1001 HTTP/1.1`

Request2: `GET http://interns-management/yash-interns/1002 HTTP/1.1`

Each of these requests can be treated separately.

A stateful design, on the other hand, looks like so:

Request1: `GET http://interns-management/yash-interns/1001 HTTP/1.1`

Request2: `GET http://interns-management/NextIntern HTTP/1.1`

To process the second request, the server needs to remember the last InternID that the client fetched. In other words, the server needs to remember the current state — otherwise Request2 cannot be processed. Design your service in a way that a request never refers to a previous request. Stateless services are easier to host, easy to maintain, and more scalable. Plus, such services can provide better response time to requests, as it is much easier to load balance them.

Links between Resources

A resource representation can contain links to other resources like an HTML page contains links to other pages. The representations returned by the service should drive the process flow as in case of a website. When you visit any website, you are presented with an index page. You click one of the links and move to another page and so on. Here, the representation is in the HTML documents and the user is driven through the website by these HTML documents themselves. The user does not need a map before coming to a website. A service can be (and should be) designed in the same manner.

Caching

Caching is the concept of storing the generated results and using the stored results instead of generating them repeatedly if the same request arrives in the near future. This can be done on the client, the server, or on any other component between them, such as a proxy server. Caching

is a great way of enhancing the service performance, but if not managed properly, it can result in client being served stale results.

Caching can be controlled using these HTTP headers:

Header	Application
`Date`	Date and time when this representation was generated.
`Last Modified`	Date and time when the server last modified this representation.
`Cache-Control`	The HTTP 1.1 header used to control caching.
`Expires`	Expiration date and time for this representation. To support HTTP 1.0 clients.
`Age`	Duration passed in seconds since this was fetched from the server. Can be inserted by an intermediary component.

Values of these headers can be used in combination with the directives in a Cache-Control header to check if the cached results are still valid or not. The most common directives for Cache-Control header are:

Directive	Application
`Public`	The default. Indicates any component can cache this representation.
`Private`	Intermediary components cannot cache this representation, only client or server can do so.
`no-cache/no-store`	Caching turned off.
`max-age`	Duration in seconds after the date-time marked in the `Date` header for which this representation is valid.
`s-maxage`	Similar to `max-age` but only meant for the intermediary caching.
`must-revalidate`	Indicates that the representation must be revalidated by the server if `max-age` has passed.
`proxy-validate`	Similar to `max-validate` but only meant for the intermediary caching.

Depending on the nature of the resources, a service can decide the values of these headers and directives. For example, a service providing stock market updates would keep the cache age limit to as low as possible or even turn off caching completely as this is a critical information and users should get the latest results all the time. On the other hand, a public picture repository whose contents do not change so frequently would use a longer caching age and slack caching rules. The server, the client, and any intermediate component between them should follow these directives to avoid outdated information getting served.

Documenting a RESTful Service

RESTful services do not necessarily require a document to help clients discover them. Due to URIs, links, and a uniform interface, it is extremely simple to discover RESTful services at runtime. A client can simply know the base address of the service and from there it can discover the service on its own by traversing through the resources using links. The method OPTION can be used effectively in the process of discovering a service.

This does not mean that RESTful services require no documentation at all. There is no excuse for not documenting your service. You should document every resource and URI for client

developers. You can use any format for structuring your document, but it should contain enough information about resources.

Document should include (for each service method):

1. HTTP Method
2. URI
3. Accept and Content-Type HTTP Request Headers
4. All possible HTTP Response codes
5. Any custom Headers
6. Sample Response
7. Sample request body for PUT, POST requests
8. Schema (xsd files for each request and response)

Error Handling

1. Service should stick to HTTP Status Codes for communicating success/failure that can be used by program clients. At times it might be necessary to add Custom HTTP Status Codes for very specific use case (these should be an exception and kept to a minimum--to a very few). Any contextual information about the error should be included in reason phrase of HTTP response or custom HTTP response headers.
2. Service requests involving user entered data (POST, PUT) should additionally include a list of user error messages in the response body that can be used to clearly display to the human users the action needed to correct the error condition. Here I also like to add a new custom HTTP Status code (450) that clearly communicates to the client that it's user data validation error.

JAX-RS Specification is the Java API for RESTful web services. JAX-RS specification is the outcome of the Java Specification Request (JSR) 311, 339. JAX-RS uses the declarative style of programming using annotations. JAX-RS provides high level simpler API to write RESTful web services that can run on Java EE and SE platforms.

The goals of the **JAX-RS API**:

POJO-based: The API will provide a set of annotations and associated classes/interfaces that may be used with POJOs in order to expose them as Web resources. The specification will define object lifecycle and scope.

HTTP-centric: The specification will assume HTTP[4] is the underlying network protocol and will provide a clear mapping between HTTP and URI[5] elements and the corresponding API classes and annotations. The API will provide high level support for common HTTP usage patterns and will be sufficiently flexible to support a variety of HTTP applications including WebDAV[6] and the Atom Publishing Protocol[7].

Format independence: The API will be applicable to a wide variety of HTTP entity body content types. It will provide the necessary pluggability to allow additional types to be added by an application in a standard manner.

Container independence: Artifacts using the API will be deployable in a variety of Web-tier containers. The specification will define how artifacts are deployed in a Servlet[8] container and as a JAX-WS[9] Provider.

Inclusion in Java EE The specification will define the environment for a Web resource class hosted in a Java EE container and will specify how to use Java EE features and components within a Web resource class.

Non-Goals of JAX-RS

Support for Java versions prior to J2SE 6.0 The API will make extensive use of annotations and will require J2SE 6.0 or later.

Description, registration and discovery The specification will neither define nor require any service description, registration or discovery capability.

HTTP Stack The specification will not define a new HTTP stack. HTTP protocol support is provided by a container that hosts artifacts developed using the API.

Data model/format classes The API will not define classes that support manipulation of entity body content, rather it will provide pluggability to allow such classes to be used by artifacts developed using the API.

Terminology

Resource class A Java class that uses JAX-RS annotations to implement a corresponding Web resource

Root resource class A resource class annotated with @Path. Root resource classes provide the roots of the resource class tree and provide access to sub-resources.

Request method designator A runtime annotation annotated with @HttpMethod. Used to identify the HTTP request method to be handled by a resource method.

Resource method A method of a resource class annotated with a request method designator that is used to handle requests on the corresponding resource.

Sub-resource locator A method of a resource class that is used to locate sub-resources of the corresponding resource.

Sub-resource method A method of a resource class that is used to handle requests on a sub-resource of the corresponding resource.

Provider An implementation of a JAX-RS extension interface. Providers extend the capabilities of a JAXRS runtime.

Filter A provider used for filtering request and responses. Entity Interceptor A provider used for intercepting calls to message body readers and writers.

Invocation A Client API object that can be configured to issue an HTTP request.

WebTarget The recipient of an Invocation, identified by a URI.

Link A URI with additional meta-data such as a media type, a relation, a title, etc.

Jersey

Jersey is the open source reference implementation of Java JAX-RS specification. It provides a Java library using which we can easily create RESTful web services in Java platform. JAX-RS / Jersey supports JAXB based XML bindings. JAXB provides API to access and process XML document.

Jersey framework is more than the JAX-RS Reference Implementation. Jersey provides it's own API that extend the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development. Jersey also exposes numerous extension SPIs so that developers may extend Jersey to best suit their needs.s

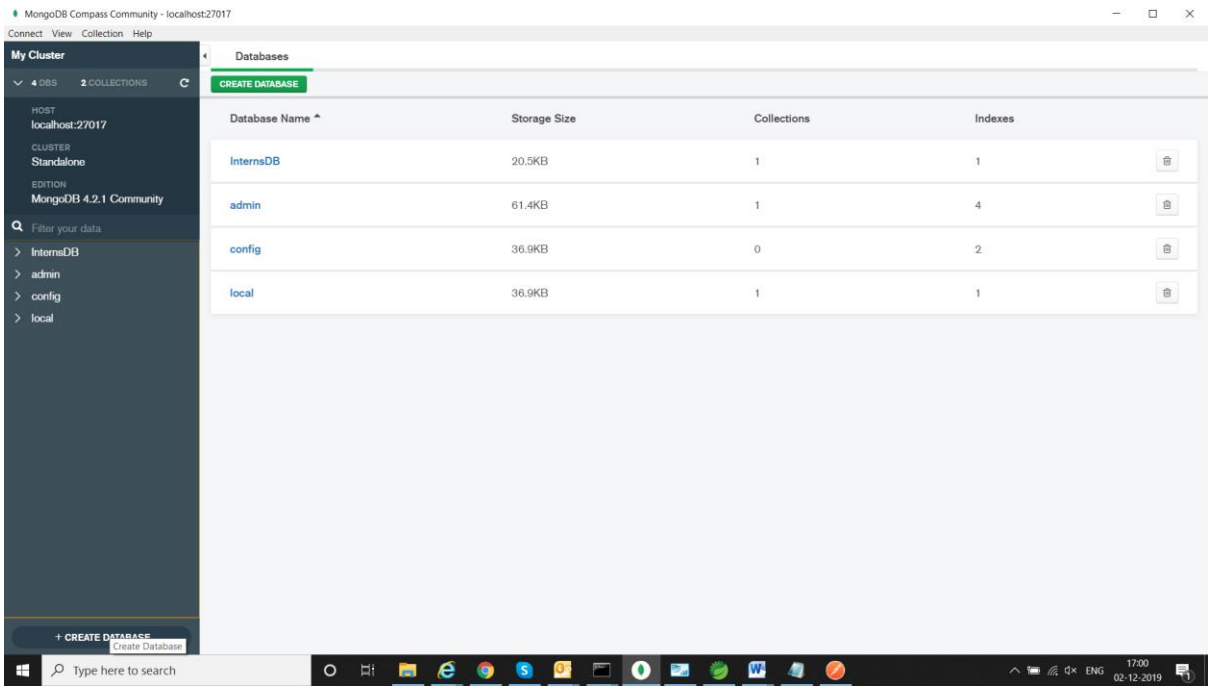
Goals of Jersey project can be summarized in the following points:

- Track the JAX-RS API and provide regular releases of production quality Reference Implementations that ships with GlassFish;
- Provide APIs to extend Jersey & Build a community of users and developers; and finally
- Make it easy to build RESTful Web services utilising Java and the Java Virtual Machine.

We will see about creating a RESTful web service using Jersey to produce JSON response. A RESTful web service can be configured to produce different types of response like XML, JSON, html, plain text and most of the MIME types.

MongoDB is a document-based NoSQL database, providing high performance and high availability. We will work with NoSQL database MongoDB.

Start Mongo DB Compass Community and create a new database InternsDB



Create a collection in InternsDB with name “Interns”

Within collection Interns add an document (similar to database row) with following details,

Let Object Id be as it is.

id:1001

internFirstName:"Sabbir"

internLastName:"Poonawala"

internAge:34

level:"ADVANCED"

Collection can also be created using below commands on command prompt.

Open CMD and type below path (installation directory of MongoDB)

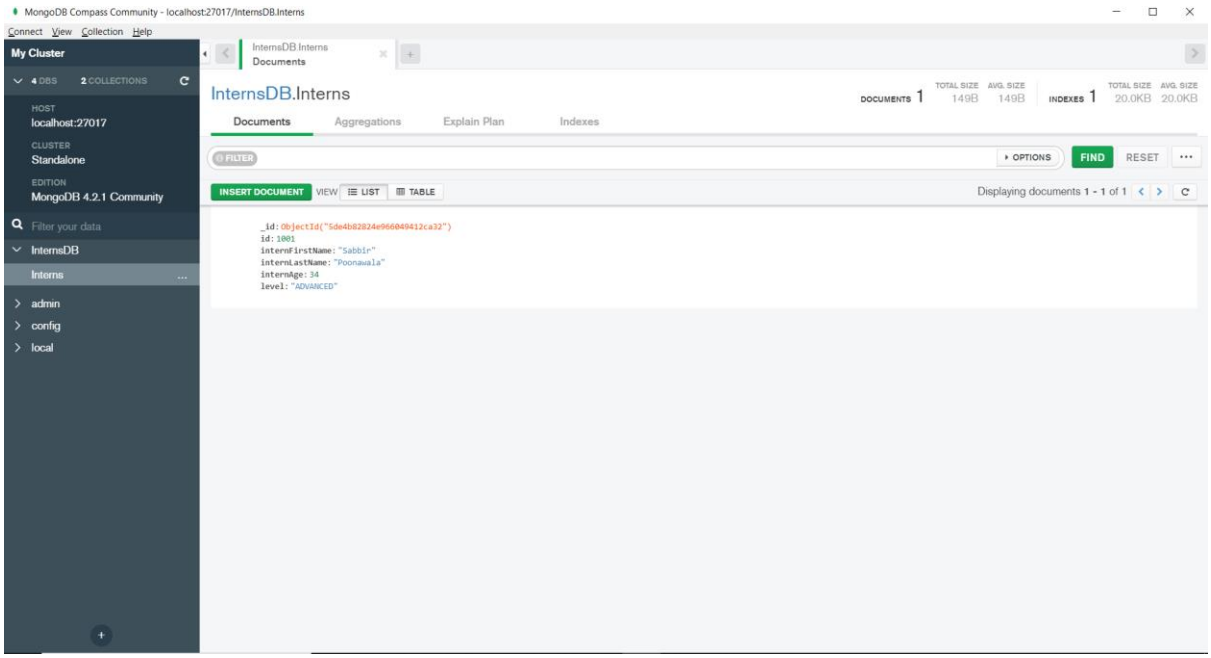
cd C:\Program Files\MongoDB\Server\4.2\bin

```
db.createCollection('Interns');
```

Once we have created the **Interns** Collection, we need to add some intern data into the collection, each of them is called as **Document** and contains the **id, internFirstName,**

internLastName, internAge and **level** of the intern. Just execute the below command to insert 1 intern document.

```
db.Interns.insert(
                {id: 1001,internFirstName:”amit", internLastName:
"desai",”internAge”:21,”level”:”ADVANCED”});
```

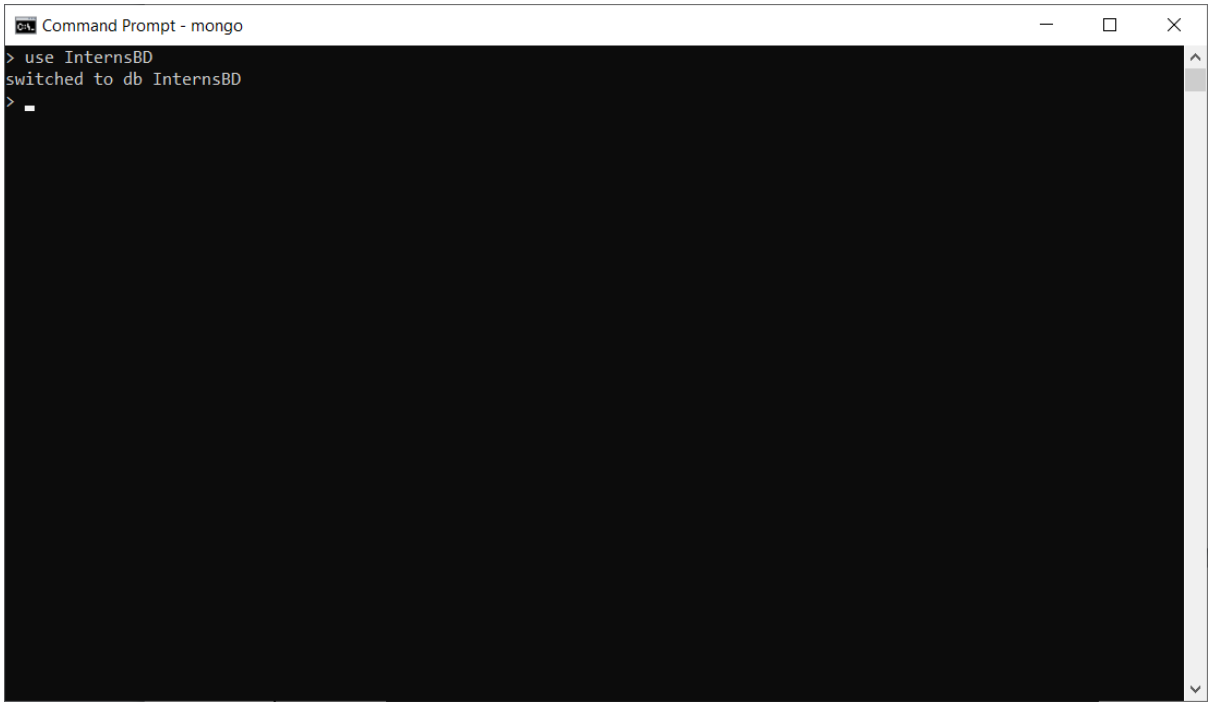


To create a user for access to InternsDB database,

Open CMD and type below path (installation directory of MongoDB)

cd C:\Program Files\MongoDB\Server\4.2\bin

Type : use InternsDB



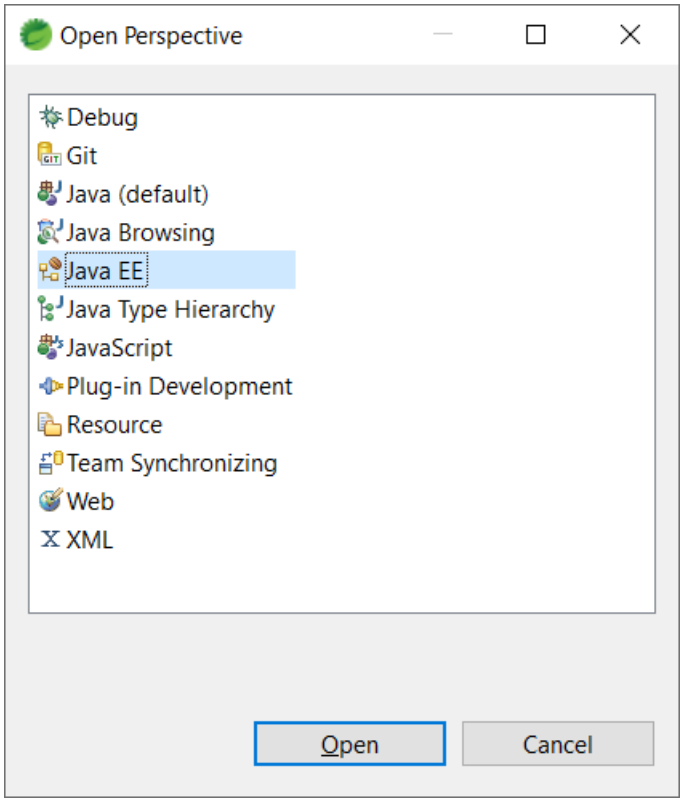
Copy below command,

```
db.createUser(
  {
    user: "sabbir",
    pwd: passwordPrompt(), // or cleartext password
    roles: []
  }
)
```

On prompting for password, enter password as “sabbir”.

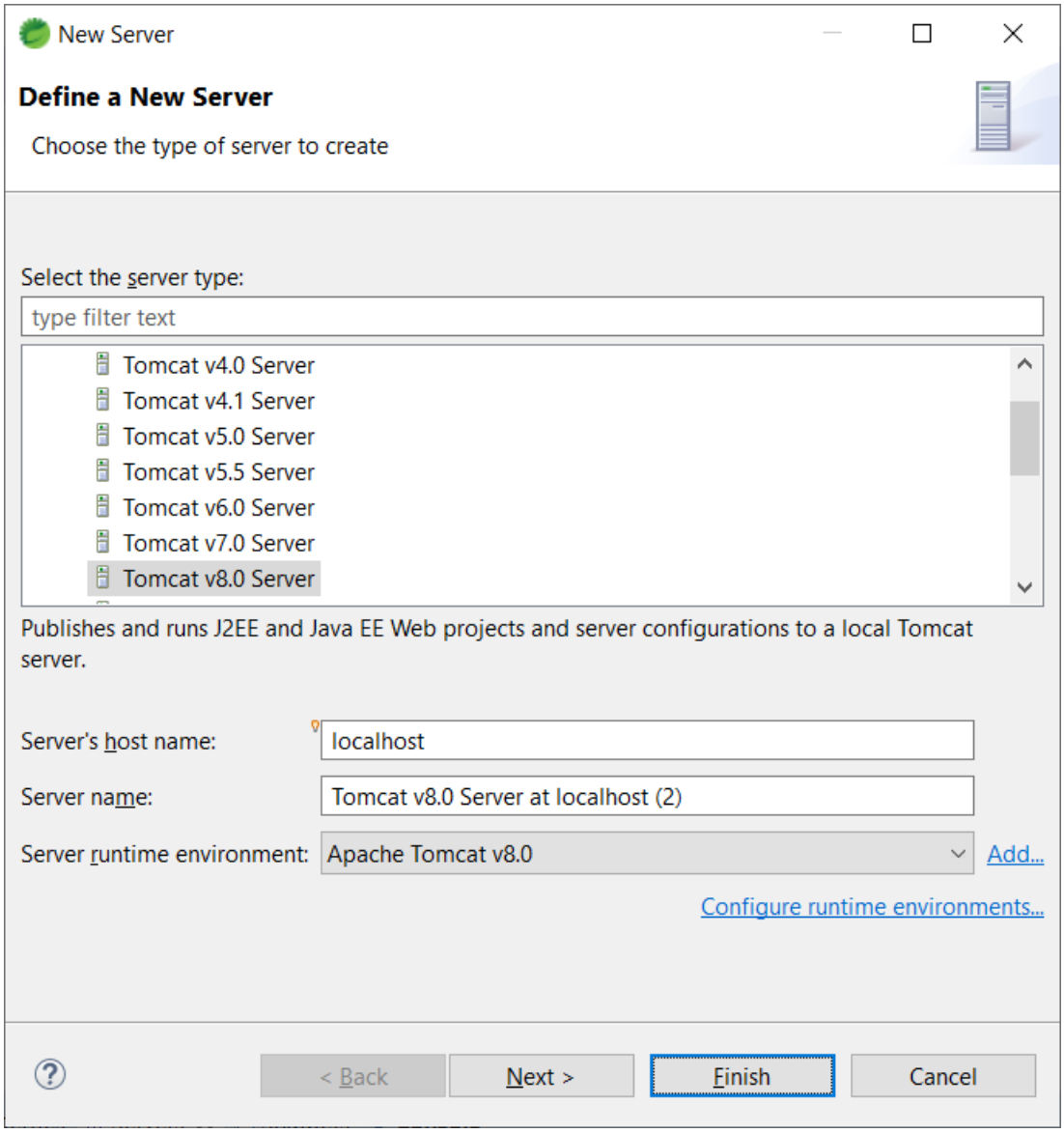
We will create Dynamic Web project in eclipse and configure it as maven project later.

In eclipse go to window menu→perspective→open perspective→other→Java EE



Go to Servers tab→click on link to add server

Select apache tomcat 8.0



On next, browse tomcat installation directory which should be ideally,

C:\Program Files\Apache Software Foundation\Tomcat 8.0

Right click on tomcat and start.

Create a new dynamic web project InternsBusinessApp Selecting runtime as Apache tomcat 8.0.

Dependencies

If you are using Glassfish application server, you don't need to package anything with your application, everything is already included. You just need to declare (provided) dependency on JAX-RS API to be able to compile your application.

Since we will use tomcat server,

Jersey core libraries,

jersey-client	Jersey core client implementation
jersey-common	Jersey core common packages
jersey-server	Jersey core server implementation

To generate POM.xml to mention dependencies of our project,

Right click IntenrsBusinessApp → configure→Maven project

Below is complete listing of pom.xml,

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>InternsBusinessApp</groupId>
  <artifactId>InternsBusinessApp</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <name>InternsBusinessApp</name>
  <description>Business application for Interns Management</description>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.2.1</version>
        <configuration>
          <warSourceDirectory>WebContent</warSourceDirectory>
        </configuration>
      </plugin>
    </plugins>
  </build>
   <properties>
    <jersey.version>1.19</jersey.version>
  </properties>
      <dependencies>
       <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-servlet</artifactId>
    <version>${jersey.version}</version>
   </dependency>
   <dependency>
    <groupId>com.sun.jersey</groupId>
```

```
     <artifactId>jersey-json</artifactId>
     <version>${jersey.version}</version>
    </dependency>
    <dependency>
        <groupId>com.sun.jersey</groupId>
        <artifactId>jersey-client</artifactId>
        <version>${jersey.version}</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>bson</artifactId>
    <version>3.11.2</version>
</dependency>
             <dependency>
       <groupId>org.mongodb</groupId>
       <artifactId>mongo-java-driver</artifactId>
       <version>3.6.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.mongodb/mongodb-driver-core -->
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-core</artifactId>
    <version>3.0.1</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.mongodb/mongodb-driver -->
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver</artifactId>
    <version>3.0.1</version>
</dependency>
       </dependencies>

</project>
```

We have added dependencies for Jersey as implementation of JAX-RS specification and jars for interacting with MongoDB database.

Jersey Configuration

A JAX-RS application is packaged as a Web application in a .war file. The application classes are packaged in WEB-INF/classes or WEB-INF/lib and required libraries are packaged in WEB-INF/lib

It is RECOMMENDED that implementations support the Servlet 3 framework pluggability mechanism to enable portability between containers and to avail themselves of container-supplied class scanning facilities.

ServletContainer is a Servlet or Filter for deploying root resource classes.

The servlet or filter may be configured to have an initialization parameter **ServletProperties.JAXRS_APPLICATION_CLASS** and whose value is a fully qualified name of a class that implements **Application**. The class is instantiated as a singleton component managed by the runtime, and injection may be performed (the artifacts that may be injected are limited to injectable providers registered when the servlet or filter is configured).

If the initialization parameter is not present and a initialization parameter "jersey.config.server.provider.packages" is present . A new instance of **ResourceConfig** with this configuration is created. The initialization parameter "jersey.config.server.provider.packages" MUST be set to provide one or more package names. Each package name MUST be separated by ';'.

If none of the above resource configuration related initialization parameters are present a new instance of **ResourceConfig** with **WebAppResourcesScanner** is created. The initialization parameter "jersey.config.server.provider.classpath" is present

ServerProperties.PROVIDER_CLASSPATH MAY be set to provide one or more resource paths. Each path MUST be separated by ';'. If the initialization parameter is not present then the following resource paths are utilized: "/WEB-INF/lib" and "/WEB-INF/classes".

All initialization parameters are added as properties of the created **ResourceConfig**.

A new **ApplicationHandler** instance will be created and configured such that the following classes may be injected onto a root resource, provider and Application classes using **@Context** annotation: HttpServletRequest, HttpServletResponse, **ServletContext**, **ServletConfig** and **WebConfig**. If this class is used as a Servlet then the **ServletConfig** class may be injected. If this class is used as a servlet filter then the **FilterConfig** class may be injected. **WebConfig** may be injected to abstract servlet or filter deployment.

In order for your service to automatically marshal and unmarshal Java Objets to and from Json you have to specify a special parameter to your Jersey serlvet configuration (obviously this will be in the web.xml file). This parameter is **com.sun.jersey.api.json.POJOMappingFeature** and will basically integrate Jersey with Jackson. **Jackson** is a high-performance JSON processor for Java. Its developers extol the combination of fast, correct, lightweight, and ergonomic attributes of the library.

Specify below configuration in web.xml,

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID" version="3.1">
  <display-name>InternsBusinessApp</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <!-- Jersey Servlet configurations -->
      <servlet>
            <servlet-name>Jersey REST Service</servlet-name>
            <servlet-
class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
            <init-param>
```

```
                    <param-name>com.sun.jersey.config.property.packages</param-
name>
                    <param-value>com.yash.controller</param-value>
                </init-param>
                <init-param>
                <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
                <param-value>true</param-value>
        </init-param>
                <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
                <servlet-name>Jersey REST Service</servlet-name>
                <url-pattern>/rest/*</url-pattern>
        </servlet-mapping>
        <!-- Jersey Servlet configurations -->

</web-app>
```

Create a package com.yash.helper and create an enum to specify constants for level of Intern

```
package com.yash.helper;
public enum Levels {
        BEGINNER(0),INTERMEDIATE(1),ADVANCED(2);
        private int level;
        private Levels(int level){
                this.level=level;
        }
        public int getLevel(){
                return level;
        }
}
```

Create a class ConnectionManager in com.yash.helper to open connection to MongoDB database

```
package com.yash.helper;
import java.net.UnknownHostException;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
import com.mongodb.ServerAddress;
import com.mongodb.client.MongoDatabase;
public class ConnectionManager {
        public static MongoDatabase getDBConnection() throws UnknownHostException {
                ServerAddress serverAddress = new ServerAddress("localhost", 27017);
                 MongoCredential
credential=MongoCredential.createCredential("sabbir", "InternsDB",
                   "sabbir".toCharArray());
                 MongoClient mongo = new MongoClient(serverAddress);
               MongoDatabase database = mongo.getDatabase("InternsDB");
               mongo.close();
    return database;
        }
}
```

Create a business entity Interns in com.yash.entity package

```
package com.yash.entity;
import java.io.Serializable;
import com.yash.helper.Levels;
public class Interns implements Serializable {
	private String objectId;
	private int id;
	private String internFirstName;
	private String internLastName;
	private int internAge;
	private Levels level;
	private int semester1Marks;
	private int semester2Marks;

	private int semester3Marks;

	public String getObjectId() {
		return objectId;
	}
	public void setObjectId(String objectId) {
		this.objectId = objectId;
	}
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getInternFirstName() {
		return internFirstName;
	}
	public void setInternFirstName(String internFirstName) {
		this.internFirstName = internFirstName;
	}
	public String getInternLastName() {
		return internLastName;
	}
	public void setInternLastName(String internLastName) {
		this.internLastName = internLastName;
	}
	public int getInternAge() {
		return internAge;
	}
	public void setInternAge(int internAge) {
		this.internAge = internAge;
	}
	public Levels getLevel() {
		return level;
	}
	public void setLevel(Levels level) {
		this.level = level;
	}
	public int getSemester1Marks() {
		return semester1Marks;
	}
	public void setSemester1Marks(int semester1Marks) {
```

```
		this.semester1Marks = semester1Marks;
	}
	public int getSemester2Marks() {
		return semester2Marks;
	}
	public void setSemester2Marks(int semester2Marks) {
		this.semester2Marks = semester2Marks;
	}
	public int getSemester3Marks() {
		return semester3Marks;
	}
	public void setSemester3Marks(int semester3Marks) {
		this.semester3Marks = semester3Marks;
	}
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + id;
		result = prime * result + internAge;
		result = prime * result + ((internFirstName == null) ? 0 :
internFirstName.hashCode());
		result = prime * result + ((internLastName == null) ? 0 :
internLastName.hashCode());
		result = prime * result + ((level == null) ? 0 : level.hashCode());
		result = prime * result + semester1Marks;
		result = prime * result + semester2Marks;
		result = prime * result + semester3Marks;
		return result;
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Interns other = (Interns) obj;
		if (id != other.id)
			return false;
		if (internAge != other.internAge)
			return false;
		if (internFirstName == null) {
			if (other.internFirstName != null)
				return false;
		} else if (!internFirstName.equals(other.internFirstName))
			return false;
		if (internLastName == null) {
			if (other.internLastName != null)
				return false;
		} else if (!internLastName.equals(other.internLastName))
			return false;
		if (level != other.level)
			return false;
		if (semester1Marks != other.semester1Marks)
			return false;
		if (semester2Marks != other.semester2Marks)
			return false;
```

```
            if (semester3Marks != other.semester3Marks)
                  return false;
            return true;
      }
      @Override
      public String toString() {
            return "Interns [id=" + id + ", internFirstName=" + internFirstName
+ ", internLastName=" + internLastName
                        + ", internAge=" + internAge + ", level=" + level + ",
semester1Marks=" + semester1Marks
                        + ", semester2Marks=" + semester2Marks + ",
semester3Marks=" + semester3Marks + "]";
      }
}
```

Create InternsDAO interface to declare data operations for business entity Interns in package com.yash.dao,

```
package com.yash.dao;
import java.net.UnknownHostException;
import java.util.List;
import com.yash.entity.Interns;
public interface InternsDAO {
      List<Interns> getAllInterns()throws UnknownHostException;
      Interns getInternById(int internId) throws UnknownHostException ;
      boolean storeInternData(Interns intern) throws UnknownHostException;
      boolean updateIntern(Interns intern)throws UnknownHostException;
      boolean updateInternLevel(Interns intern)throws UnknownHostException;
      boolean removeIntern(int internId)throws UnknownHostException;
}
```

And implementation class InternsDAOImpl as below,

```
package com.yash.dao;

import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
import org.bson.Document;
import com.mongodb.BasicDBObject;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;
import com.yash.entity.Interns;
import com.yash.helper.ConnectionManager;
import com.yash.helper.Levels;

public class InternsDAOImpl implements InternsDAO {
      @Override
      public List<Interns> getAllInterns() throws UnknownHostException {
            // TODO Auto-generated method stub
            MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
```

```
            MongoCollection<Document> internsCollection =
mongoDatabase.getCollection("Interns");
            FindIterable<Document> documentIterable=internsCollection.find();
            MongoCursor<Document> cursor=documentIterable.iterator();
            List<Interns> internsList=new ArrayList<Interns>();
            while(cursor.hasNext()) {
                  Document document=cursor.next();
                  Interns interns=new Interns();
                  interns.setId((Integer)document.get("id"));
      interns.setInternFirstName((String)document.get("internFirstName"));
      interns.setInternLastName((String)document.get("internLastName"));
                  interns.setInternAge((Integer)document.get("internAge"));
                  switch((String)document.get("level")) {
                  case "ADVANCED":
                                    interns.setLevel(Levels.ADVANCED);
                                    break;
                  case "BEGINNER":
                                    interns.setLevel(Levels.BEGINNER);
                                    break;
                  case "INTERMEDIATE":
                                    interns.setLevel(Levels.INTERMEDIATE);
                                    break;
                  }
                  internsList.add(interns);
            }

            return internsList;
      }

      @Override
      public Interns getInternById(int internId) throws UnknownHostException {
            // TODO Auto-generated method stub
            BasicDBObject searchQuery = new BasicDBObject();
            searchQuery.put("id", internId);
            MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
            MongoCollection<Document> internsCollection =
mongoDatabase.getCollection("Interns");
            FindIterable<Document>
documentIterable=internsCollection.find(Filters.eq("id", internId));
            MongoCursor<Document> cursor=documentIterable.iterator();
            Interns interns=new Interns();
            while (cursor.hasNext()) {
                  Document document= cursor.next();
                  interns.setId((Integer)document.get("id"));
      interns.setInternFirstName((String)document.get("internFirstName"));
      interns.setInternLastName((String)document.get("internLastName"));
                  interns.setInternAge((Integer)document.get("internAge"));
                  switch((String)document.get("level")) {
                  case "ADVANCED":
                                    interns.setLevel(Levels.ADVANCED);
                                    break;
                  case "BEGINNER":
                                    interns.setLevel(Levels.BEGINNER);
                                    break;
                  case "INTERMEDIATE":
```

```
                                        interns.setLevel(Levels.INTERMEDIATE);
                                        break;
                                }
                        }
                        return interns;
                }

                @Override
                public boolean storeInternData(Interns intern) throws UnknownHostException
{
                        // TODO Auto-generated method stub
                        MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
                Document document=new Document()
                                        .append("id", intern.getId())
                                        .append("internFirstName",intern.getInternFirstName())
                                        .append("internLastName",intern.getInternLastName())
                                        .append("internAge",intern.getInternAge())
                                        .append("level",intern.getLevel().toString());
                                    MongoCollection<Document> internsCollection =
mongoDatabase.getCollection("Interns");
                                    internsCollection.insertOne(document);
                                return true;
                }
                @Override
                public boolean updateIntern(Interns intern) throws UnknownHostException {
                        // TODO Auto-generated method stub
                        MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
                        MongoCollection<Document>
internsCollection=mongoDatabase.getCollection("Interns");
                        Document document=new Document()
                                        .append("id", intern.getId())
                                        .append("internFirstName",intern.getInternFirstName())
                                        .append("internLastName",intern.getInternLastName())
                                        .append("internAge",intern.getInternAge())
                                        .append("level",intern.getLevel().name());
                        System.out.println("id:"+intern.getId());
                                Document
documentDB=internsCollection.findOneAndReplace(Filters.eq("id",intern.getId()),
document);
                System.out.println(documentDB);
                        return true;
                }
                @Override
                public boolean updateInternLevel(Interns intern) throws
UnknownHostException {
                        // TODO Auto-generated method stub
                        MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
                        MongoCollection<Document>
internsCollection=mongoDatabase.getCollection("Interns");
                        internsCollection.updateOne(Filters.eq("id", intern.getId()),
Updates.set("level", intern.getLevel().name()));
                        return true;
                }

                @Override
                public boolean removeIntern(int internId) throws UnknownHostException {
                        // TODO Auto-generated method stub
                        MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
```

```
            MongoCollection<Document>
internsCollection=mongoDatabase.getCollection("Interns");
            internsCollection.deleteOne(Filters.eq("id",internId));
            return true;
      }

}
```

All CRUD-related methods in the Java driver are accessed through the **MongoCollection** interface. Instances of **MongoCollection** can be obtained from a **MongoClient** instance by way of a **MongoDatabase**.

```
MongoClient client = new MongoClient();
MongoDatabase database = client.getDatabase("InternsDB");
MongoCollection<Document> collection = database.getCollection("interns");
```

MongoCollection is a generic interface: the TDocument type parameter is the class that clients use to insert or replace documents in a collection, and the default type returned from find and aggregate.

The single-argument getCollection method returns an instance of **MongoCollection<Document>**, and so with this type of collection an application uses instances of the **Document** class.

The complete listing of InternsDAOImpl is as below,

```
package com.yash.dao;

import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
import org.bson.Document;
import com.mongodb.BasicDBObject;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;
import com.yash.entity.Interns;
import com.yash.helper.ConnectionManager;
import com.yash.helper.Levels;
public class InternsDAOImpl implements InternsDAO {
      @Override
      public List<Interns> getAllInterns() throws UnknownHostException {
            // TODO Auto-generated method stub
            MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
            MongoCollection<Document> internsCollection =
mongoDatabase.getCollection("Interns");
            FindIterable<Document> documentIterable=internsCollection.find();
            MongoCursor<Document> cursor=documentIterable.iterator();
```

```
            List<Interns> internsList=new ArrayList<Interns>();
            while(cursor.hasNext()) {
                  Document document=cursor.next();
                  Interns interns=new Interns();
                  interns.setId((Integer)document.get("id"));
      interns.setInternFirstName((String)document.get("internFirstName"));
      interns.setInternLastName((String)document.get("internLastName"));
                  interns.setInternAge((Integer)document.get("internAge"));
                  switch((String)document.get("level")) {
                  case "ADVANCED":
                                    interns.setLevel(Levels.ADVANCED);
                                    break;
                  case "BEGINNER":
                                    interns.setLevel(Levels.BEGINNER);
                                    break;
                  case "INTERMEDIATE":
                                    interns.setLevel(Levels.INTERMEDIATE);
                                    break;
                  }
                  internsList.add(interns);
            }

            return internsList;
      }

      @Override
      public Interns getInternById(int internId) throws UnknownHostException {
            // TODO Auto-generated method stub
            BasicDBObject searchQuery = new BasicDBObject();
            searchQuery.put("id", internId);
            MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
            MongoCollection<Document> internsCollection =
mongoDatabase.getCollection("Interns");
            FindIterable<Document>
documentIterable=internsCollection.find(Filters.eq("id", internId));
            MongoCursor<Document> cursor=documentIterable.iterator();
            Interns interns=new Interns();
            while (cursor.hasNext()) {
                  Document document= cursor.next();
                  interns.setId((Integer)document.get("id"));
      interns.setInternFirstName((String)document.get("internFirstName"));
      interns.setInternLastName((String)document.get("internLastName"));
                  interns.setInternAge((Integer)document.get("internAge"));
                  switch((String)document.get("level")) {
                  case "ADVANCED":
                                    interns.setLevel(Levels.ADVANCED);
                                    break;
                  case "BEGINNER":
                                    interns.setLevel(Levels.BEGINNER);
                                    break;
                  case "INTERMEDIATE":
                                    interns.setLevel(Levels.INTERMEDIATE);
                                    break;
                  }
            }
```

```
            return interns;
      }

      @Override
      public boolean storeInternData(Interns intern) throws UnknownHostException
{
            // TODO Auto-generated method stub
            MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
      Document document=new Document()
                        .append("id", intern.getId())
                        .append("internFirstName",intern.getInternFirstName())
                        .append("internLastName",intern.getInternLastName())
                        .append("internAge",intern.getInternAge())
                        .append("level",intern.getLevel().toString());
                       MongoCollection<Document> internsCollection =
mongoDatabase.getCollection("Interns");
                       internsCollection.insertOne(document);
                     return true;
      }
      @Override
      public boolean updateIntern(Interns intern) throws UnknownHostException {
            // TODO Auto-generated method stub
            MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
            MongoCollection<Document>
internsCollection=mongoDatabase.getCollection("Interns");
            Document document=new Document()
                        .append("id", intern.getId())
                        .append("internFirstName",intern.getInternFirstName())
                        .append("internLastName",intern.getInternLastName())
                        .append("internAge",intern.getInternAge())
                        .append("level",intern.getLevel().name());
            System.out.println("id:"+intern.getId());
                     Document
documentDB=internsCollection.findOneAndReplace(Filters.eq("id",intern.getId()),
document);
      System.out.println(documentDB);
            return true;
      }
      @Override
      public boolean updateInternLevel(Interns intern) throws
UnknownHostException {
            // TODO Auto-generated method stub
            MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
            MongoCollection<Document>
internsCollection=mongoDatabase.getCollection("Interns");
            internsCollection.updateOne(Filters.eq("id", intern.getId()),
Updates.set("level", intern.getLevel().name()));
            return true;
      }

      @Override
      public boolean removeIntern(int internId) throws UnknownHostException {
            // TODO Auto-generated method stub
            MongoDatabase mongoDatabase=ConnectionManager.getDBConnection();
            MongoCollection<Document>
internsCollection=mongoDatabase.getCollection("Interns");
            internsCollection.deleteOne(Filters.eq("id",internId));
            return true;
      }
```

```
}
```

Create a business exception in package com.yash.exception,

```
package com.yash.exception;
public class InternsException extends RuntimeException{
	private String message;
	public InternsException(String message) {
		this.message=message;
	}
	public String getMessage() {
		return message;
	}
}
```

Business Service Interface InternsService and implementation class in InternsServiceImpl in package com.yash.service.

```
package com.yash.service;
import java.util.List;
import com.yash.entity.Interns;
public interface InternsService {
	List<Interns> retrieveInternsService();
	Interns retrieveInternsByIdService(int internId);
	boolean registerInternService(Interns interns);
	boolean updateInternService(Interns interns);
	boolean updateInternLevelService(Interns interns);
	boolean removeInternService(int internId);

}
```

To hide the implementation details of DAO and Service interfaces, we will create Factory class in com.yash.helper package.

```
package com.yash.helper;
import com.yash.dao.InternsDAO;
import com.yash.dao.InternsDAOImpl;
import com.yash.service.InternsService;
import com.yash.service.InternsServiceImpl;
public class FactoryInterns {
	public static InternsDAO createInternsDAO() {
		InternsDAO internsDAO=new InternsDAOImpl();
		return internsDAO;
	}
	public static InternsService createInternsService() {
		InternsService internsService=new InternsServiceImpl();
		return internsService;
	}
```

```
}
```

Business Interface implementation class in com.yash.service package,

```
package com.yash.service;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
import com.yash.dao.InternsDAO;
import com.yash.entity.Interns;
import com.yash.exception.InternsException;
import com.yash.helper.FactoryInterns;
import com.yash.helper.Levels;
public class InternsServiceImpl implements InternsService{
	InternsDAO internsDAO=FactoryInterns.createInternsDAO();
	@Override
	public List<Interns> retrieveInternsService() {
		// TODO Auto-generated method stub
		List<Interns> internsList=new ArrayList<Interns>();
		try {
			internsList=internsDAO.getAllInterns();
		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		return internsList;
	}

	@Override
	public Interns retrieveInternsByIdService(int internId) {
		// TODO Auto-generated method stub
		Interns interns=null;
		try {
			interns = internsDAO.getInternById(internId);
		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		return interns;
	}
	public Levels determineLevelBySemesterMarks(Interns interns){
		int sem1Marks=interns.getSemester1Marks();
		int sem2Marks=interns.getSemester2Marks();
		int sem3Marks=interns.getSemester3Marks();
		int semAverage=(sem1Marks+sem2Marks+sem3Marks)/3;
		if(semAverage<=50){
			throw new InternsException("Intern did not match
eligibility");
		}
		if(semAverage>50 && semAverage<=60){
			return Levels.BEGINNER;
		}else if(semAverage>60 && semAverage<70){
			return Levels.INTERMEDIATE;
		}else{
			return Levels.ADVANCED;
		}
	}
```

```
	@Override
	public boolean registerInternService(Interns interns) {
		// TODO Auto-generated method stub
		boolean internRegistered=false;
		Levels levels=determineLevelBySemesterMarks(interns);
		interns.setLevel(levels);
		try {
			internRegistered=internsDAO.storeInternData(interns);
		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			throw new InternsException("Intern did not match
eligibility");
		}
		return internRegistered;
	}

	@Override
	public boolean updateInternService(Interns interns) {
		// TODO Auto-generated method stub
		Levels levels=determineLevelBySemesterMarks(interns);
		interns.setLevel(levels);
		boolean internLevelUpdated=false;
		try {
			internLevelUpdated=internsDAO.updateIntern(interns);
		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			throw new InternsException("Intern did not match
eligibility");
		}
		return internLevelUpdated;
	}

	@Override
	public boolean updateInternLevelService(Interns interns) {
		// TODO Auto-generated method stub
		boolean internLevelUpdated=false;
		Levels levels=determineLevelBySemesterMarks(interns);
		interns.setLevel(levels);
		try {
			internLevelUpdated=internsDAO.updateInternLevel(interns);
		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			throw new InternsException("Intern did not match
eligibility");
		}
		return internLevelUpdated;
	}

	@Override
	public boolean removeInternService(int internId) {
		// TODO Auto-generated method stub
		boolean internRemoved=false;
		try {
			internRemoved=internsDAO.removeIntern(internId);
		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			throw new InternsException("Intern deletion failed");
		}
		return internRemoved;
```

```
	}
}
```

We will create two **model** beans – **InternsRequest** for our **request** data and InternsResponse for sending response to client systems. For sending XML response, the beans should be annotated with **@XmlRootElement**. Since we are sending JSON response we are not applying **@XmlRootElement**.

```
package com.yash.model;
import com.yash.helper.Levels;
public class InternsRequest {
	private int id;
	private String internFirstName;
	private String internLastName;
	private int internAge;
	private Levels level;
	private int semester1Marks;
	private int semester2Marks;
	private int semester3Marks;

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getInternFirstName() {
		return internFirstName;
	}
	public void setInternFirstName(String internFirstName) {
		this.internFirstName = internFirstName;
	}
	public String getInternLastName() {
		return internLastName;
	}
	public void setInternLastName(String internLastName) {
		this.internLastName = internLastName;
	}
	public int getInternAge() {
		return internAge;
	}
	public void setInternAge(int internAge) {
		this.internAge = internAge;
	}
	public Levels getLevel() {
		return level;
	}
	public void setLevel(Levels level) {
		this.level = level;
	}
	public int getSemester1Marks() {
```

```
		return semester1Marks;
	}
	public void setSemester1Marks(int semester1Marks) {
		this.semester1Marks = semester1Marks;
	}
	public int getSemester2Marks() {
		return semester2Marks;
	}
	public void setSemester2Marks(int semester2Marks) {
		this.semester2Marks = semester2Marks;
	}
	public int getSemester3Marks() {
		return semester3Marks;
	}
	public void setSemester3Marks(int semester3Marks) {
		this.semester3Marks = semester3Marks;
	}

}
```

And InternsResponse

```
package com.yash.model;
import com.yash.helper.Levels;
public class InternsResponse {
	private int id;
	private String internFirstName;
	private String internLastName;
	private int internAge;
	private Levels level;
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getInternFirstName() {
		return internFirstName;
	}
	public void setInternFirstName(String internFirstName) {
		this.internFirstName = internFirstName;
	}
	public String getInternLastName() {
		return internLastName;
	}
	public void setInternLastName(String internLastName) {
		this.internLastName = internLastName;
	}
	public int getInternAge() {
		return internAge;
	}
	public void setInternAge(int internAge) {
		this.internAge = internAge;
	}
	public Levels getLevel() {
		return level;
	}
	public void setLevel(Levels level) {
		this.level = level;
```

```
        }
}
```

Resource Classes

A resource class is a Java class that uses JAX-RS annotations to implement a corresponding Web resource. Resource classes are POJOs that have at least one method annotated with @Path or a request method designator.

Lifecycle and Environment By default a new resource class instance is created for each request to that resource. First the constructor is called, then any requested dependencies are injected, then the appropriate method is invoked and finally the object is made available for garbage collection. An implementation MAY offer other resource class lifecycles, mechanisms for specifying these are outside the scope of this specification. E.g. an implementation based on an inversion-of-control framework may support all of the lifecycle options provided by that framework.

Constructors Root resource classes are instantiated by the JAX-RS runtime and MUST have a public constructor for which the JAX-RS runtime can provide all parameter values. Note that a zero argument constructor is permissible under this rule. A public constructor MAY include parameters annotated with one of the following: @Context, @HeaderParam, @CookieParam, @MatrixParam, @QueryParam or @PathParam. However, depending on the resource class lifecycle and concurrency, per-request information may not make sense in a constructor. If more than one public constructor is suitable then an implementation MUST use the one with the most parameters. Choosing amongst suitable constructors with the same number of parameters is implementation specific, implementations SHOULD generate a warning about such ambiguity.

Non-root resource classes are instantiated by an application and do not require the above-described public constructor

Fields and Bean Properties

When a resource class is instantiated, the values of fields and bean properties annotated with one the following annotations are set according to the semantics of the annotation:

@MatrixParam Extracts the value of a URI matrix parameter.

@QueryParam Extracts the value of a URI query parameter.

@PathParam Extracts the value of a URI template parameter.

@CookieParam Extracts the value of a cookie.

@HeaderParam Extracts the value of a header.

@Context Injects an instance of a supported resource

@**QueryParam** and @**PathParam** can only be used on the following Java types:

- All primitive types except char
- All wrapper classes of primitive types except Character
- Have a constructor that accepts a single String argument
- Any class with the static method named valueOf(String) that accepts a single String argument
- Any class with a constructor that takes a single String as a parameter
- List<T>, Set<T>, or SortedSet<T>, where T matches the already listed criteria. Sometimes parameters may contain more than one value for the same name. If this is the case, these types may be used to obtain all values**.**

Resource Methods

Resource methods are methods of a resource class annotated with a request method designator. They are used to handle requests and MUST conform to certain restrictions.

A request method designator is a runtime annotation that is annotated with the @HttpMethod annotation. JAX-RS defines a set of request method designators for the common HTTP methods: @GET, @POST, @PUT, @DELETE, @HEAD and @OPTIONS.

Visibility

Only public methods may be exposed as resource methods. An implementation SHOULD warn users if a non-public method carries a method designator or @Path annotation.

Parameters

When a resource method is invoked, parameters annotated with @FormParam or one of the annotations listed in above are mapped from the request according to the semantics of the annotation. Similar to fields and bean properties.

The DefaultValue annotation may be used to supply a default value for parameters. The Encoded annotation may be used to disable automatic URI decoding of parameter values. Exceptions thrown during construction of parameter values are treated the same as exceptions thrown during construction of field or bean property values. Exceptions thrown during construction of @FormParam annotated parameter values are treated the same as if the parameter were annotated with @HeaderParam.

Entity Parameters

The value of a parameter not annotated with @FormParam or any of the annotations(@QueryParam, @MatrixParam etc.) listed above called the entity parameter, is mapped from the request entity body. Conversion between an entity body and a Java type is the responsibility of an entity provider. Resource methods MUST have at most one entity parameter

Return Type

Resource methods MAY return void, Response, GenericEntity, or another Java type, these return types are mapped to a response entity body as follows:

void Results in an empty entity body with a 204 status code.

Response Results in an entity body mapped from the entity property of the Response with the status code specified by the status property of the Response. A null return value results in a 204 status code. If the status property of the Response is not set: a 200 status code is used for a non-null entity property and a 204 status code is used if the entity property is null.

GenericEntity Results in an entity body mapped from the Entity property of the GenericEntity. If the return value is not null a 200 status code is used, a null return value results in a 204 status code.

Other Results in an entity body mapped from the class of the returned instance; if this class is an anonymous inner class, its superclass is used instead. If the return value is not null a 200 status code is used, a null return value results in a 204 status code.

Sub Resources

Methods of a resource class that are annotated with @Path are either sub-resource methods or sub-resource locators. Sub-resource methods handle a HTTP request directly whilst sub-resource locators return an object that will handle a HTTP request.

The presence or absence of a request method designator (e.g. @GET) differentiates between the two: Present Such methods, known as sub-resource methods, are treated like a normal resource method except the method is only invoked for request URIs that match a URI template created by concatenating the URI template of the resource class with the URI template of the method.

Absent Such methods, known as sub-resource locators, are used to dynamically resolve the object that will handle the request. Any returned object is treated as a resource class instance and used to either handle the request or to further resolve the object that will handle the request.

An implementation MUST dynamically determine the class of object returned rather than relying on the static sub-resource locator return type since the returned instance may be a subclass of the declared type with potentially different annotations.

Subresource locators may have all the same parameters as a normal resource method except that they MUST NOT have an entity parameter.

Declaring Media Type Capabilities

Resource classes can declare the supported request and response media types using the @Consumes and @Produces annotations respectively. These annotations MAY be applied to a resource method, a resource class, or to an entity provider. Use of these annotations on a resource method overrides any on the resource class or on an entity provider for a method argument or return type. In the absence of either of these annotations, support for any media type (“*/*”) is assumed.

The following list contains the standard types that are supported automatically for entities. You only need to write an entity provider if you are not choosing one of the following, standard types.

- byte[] — All media types (*/*)
- java.lang.String — All text media types (text/*)
- java.io.InputStream — All media types (*/*)
- java.io.Reader — All media types (*/*)
- java.io.File — All media types (*/*)
- javax.activation.DataSource — All media types (*/*)
- javax.xml.transform.Source — XML types (text/xml, application/xml and application/*+xml)
- javax.xml.bind.JAXBElement and application-supplied JAXB classes XML media types (text/xml, application/xml and application/*+xml)
- MultivaluedMap<String, String> — Form content (application/x-www-form-urlencoded)
- StreamingOutput — All media types (*/*), MessageBodyWriter only

An implementation MUST NOT invoke a method whose effective value of @Produces does not match the request Accept header. An implementation MUST NOT invoke a method whose effective value of @Consumes does not match the request Content-Type header.

Entity Providers

Entity providers supply mapping services between representations and their associated Java types.

For HTTP requests, the MessageBodyReader is used to map an HTTP request entity body to method parameters.

On the response side, a return value is mapped to an HTTP response entity body using a MessageBodyWriter. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a Response that wraps the entity, and which can be built using Response.ResponseBuilder.

We will create controller class as a resource class for ease of understanding Controller in Spring REST discussed in next section.

Create InternsController in com.yash.controller package

```
package com.yash.controller;
import java.util.ArrayList;
```

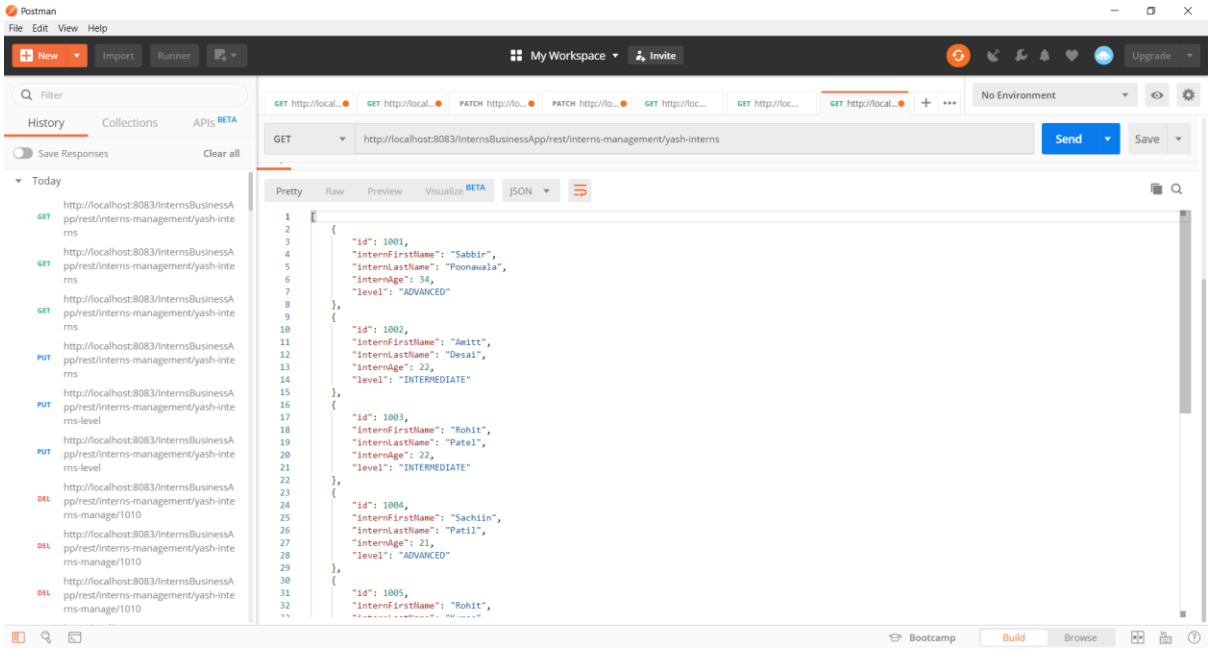
```java
import java.util.List;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import com.sun.jersey.api.client.ClientResponse.Status;
import com.yash.entity.Interns;
import com.yash.helper.FactoryInterns;
import com.yash.model.InternsRequest;
import com.yash.model.InternsResponse;
import com.yash.service.InternsService;
@Path("interns-management")
public class InternsController {
	private InternsService internsService=null;
	public InternsController() {
		this.internsService=FactoryInterns.createInternsService();
	}
	@Produces(MediaType.APPLICATION_JSON)
	@Path(value="/yash-interns")
	@GET
	public List<InternsResponse> retrieveAllInterns() {
		List<Interns> internsList=internsService.retrieveInternsService();
		List<InternsResponse> responseList=new ArrayList<InternsResponse>();
		for(Interns interns:internsList) {
			InternsResponse response=new InternsResponse();
			response.setId(interns.getId());
			response.setInternFirstName(interns.getInternFirstName());
			response.setInternLastName(interns.getInternLastName());
			response.setInternAge(interns.getInternAge());
			response.setLevel(interns.getLevel());
			responseList.add(response);
		}
	return responseList;
	}
	@Produces(MediaType.APPLICATION_JSON)
	@Path(value="/yash-interns/{id}")
	@GET
	public Response retrieveInternById(@PathParam("id") int internId) {
		Interns interns=internsService.retrieveInternsByIdService(internId);
		InternsResponse response=new InternsResponse();
		response.setId(interns.getId());
		response.setInternFirstName(interns.getInternFirstName());
		response.setInternLastName(interns.getInternLastName());
		response.setInternAge(interns.getInternAge());
		response.setLevel(interns.getLevel());
		return Response
				.status(Response.Status.OK)
				.type(MediaType.APPLICATION_JSON)
				.entity(response)
				.build();
	}
	@POST
	@Consumes(MediaType.APPLICATION_JSON)
```

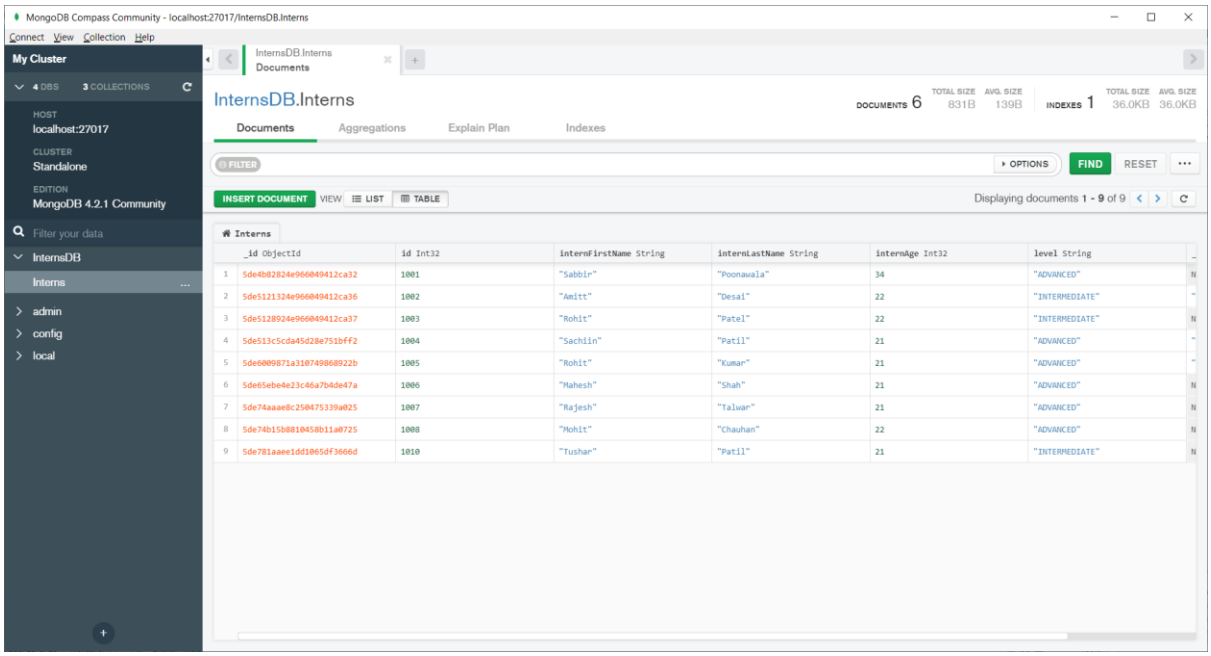
```
	@Path(value="/yash-interns")
	public Response registerInterns(InternsRequest internsRequest) {
		Interns interns=new Interns();
		interns.setId(internsRequest.getId());
		interns.setInternFirstName(internsRequest.getInternFirstName());
		interns.setInternLastName(internsRequest.getInternLastName());
		interns.setInternAge(internsRequest.getInternAge());
		interns.setSemester1Marks(internsRequest.getSemester1Marks());
		interns.setSemester2Marks(internsRequest.getSemester2Marks());
		interns.setSemester3Marks(internsRequest.getSemester3Marks());
		boolean
internRegistered=internsService.registerInternService(interns);
		if(internRegistered) {
			InternsResponse response=new InternsResponse();
			response.setId(interns.getId());
			response.setInternFirstName(interns.getInternFirstName());
			response.setInternLastName(interns.getInternLastName());
			response.setInternAge(interns.getInternAge());
			response.setLevel(interns.getLevel());
			return
Response.ok(response).type(MediaType.APPLICATION_JSON).build();

		}else {
			return Response.status(Status.CONFLICT).build();
	}
	}
	@PUT
	@Consumes(MediaType.APPLICATION_JSON)
	@Path(value="/yash-interns-manage")
	public Response updateInterns(InternsRequest internsRequest) {
		Interns interns=new Interns();
		interns.setId(internsRequest.getId());
		interns.setInternFirstName(internsRequest.getInternFirstName());
		interns.setInternLastName(internsRequest.getInternLastName());
		interns.setInternAge(internsRequest.getInternAge());
		interns.setSemester1Marks(internsRequest.getSemester1Marks());
		interns.setSemester2Marks(internsRequest.getSemester2Marks());
		interns.setSemester3Marks(internsRequest.getSemester3Marks());
		boolean internUpdated=internsService.updateInternService(interns);
		if(internUpdated) {
			InternsResponse response=new InternsResponse();
			response.setId(interns.getId());
			response.setInternFirstName(interns.getInternFirstName());
			response.setInternLastName(interns.getInternLastName());
			response.setInternAge(interns.getInternAge());
			response.setLevel(interns.getLevel());
			return
Response.ok(response).type(MediaType.APPLICATION_JSON).build();

		}else {
			return Response.status(Status.NOT_MODIFIED).build();
	}
	}
	@PUT
	@Consumes(MediaType.APPLICATION_JSON)
	@Path(value="/yash-interns-level")
	public Response updateInternsLevel(InternsRequest internsRequest) {
		Interns interns=new Interns();
		interns.setId(internsRequest.getId());
```

```
			interns.setSemester1Marks(internsRequest.getSemester1Marks());
			interns.setSemester2Marks(internsRequest.getSemester2Marks());
			interns.setSemester3Marks(internsRequest.getSemester3Marks());
			boolean
internLevelUpdated=internsService.updateInternLevelService(interns);
			if(internLevelUpdated) {
				InternsResponse response=new InternsResponse();
				response.setId(interns.getId());
				response.setLevel(interns.getLevel());
				return
Response.ok(response).type(MediaType.APPLICATION_JSON).build();

			}else {
				return Response.status(Status.NOT_MODIFIED).build();
	}
	}

	@DELETE
	@Consumes(MediaType.APPLICATION_JSON)
	@Path(value="/yash-interns-manage/{id}")
	public Response removeInternsService(@PathParam("id") int internId) {
	boolean internRemoved=internsService.removeInternService(internId);
			if(internRemoved) {
				InternsResponse response=new InternsResponse();
				response.setId(internId);
				return
Response.ok(response).type(MediaType.APPLICATION_JSON).build();

			}else {
				return Response.status(Status.NOT_ACCEPTABLE).build();

	}
	}
}
```

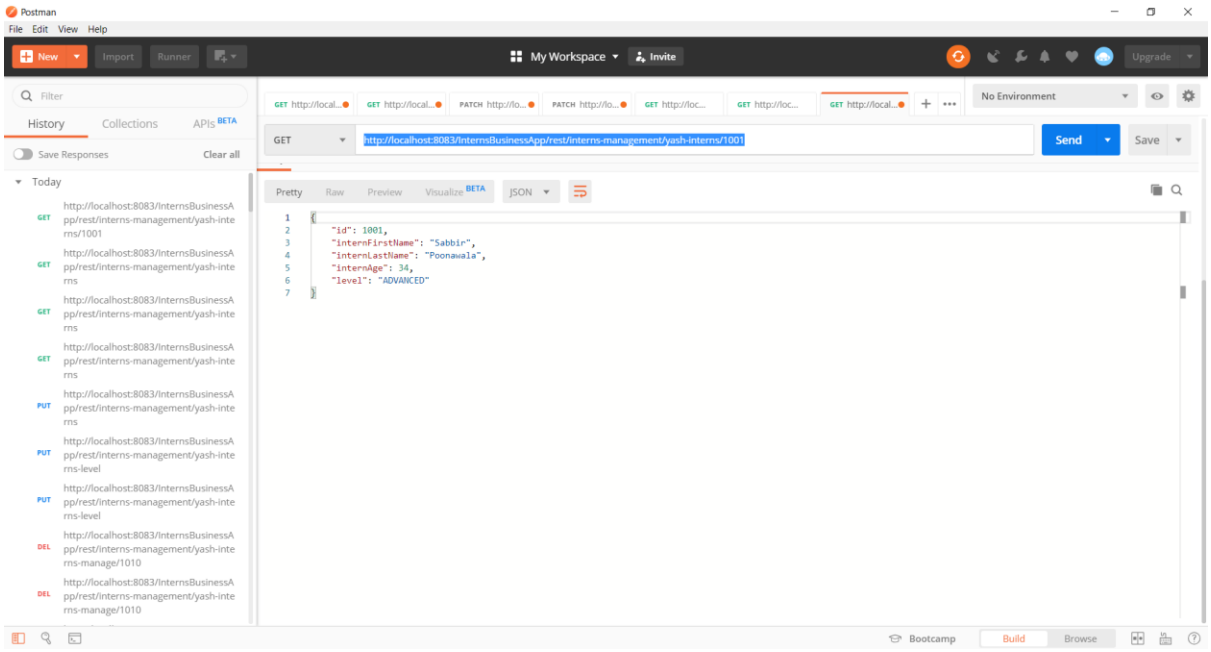
To test services using postman,

GET ： http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns

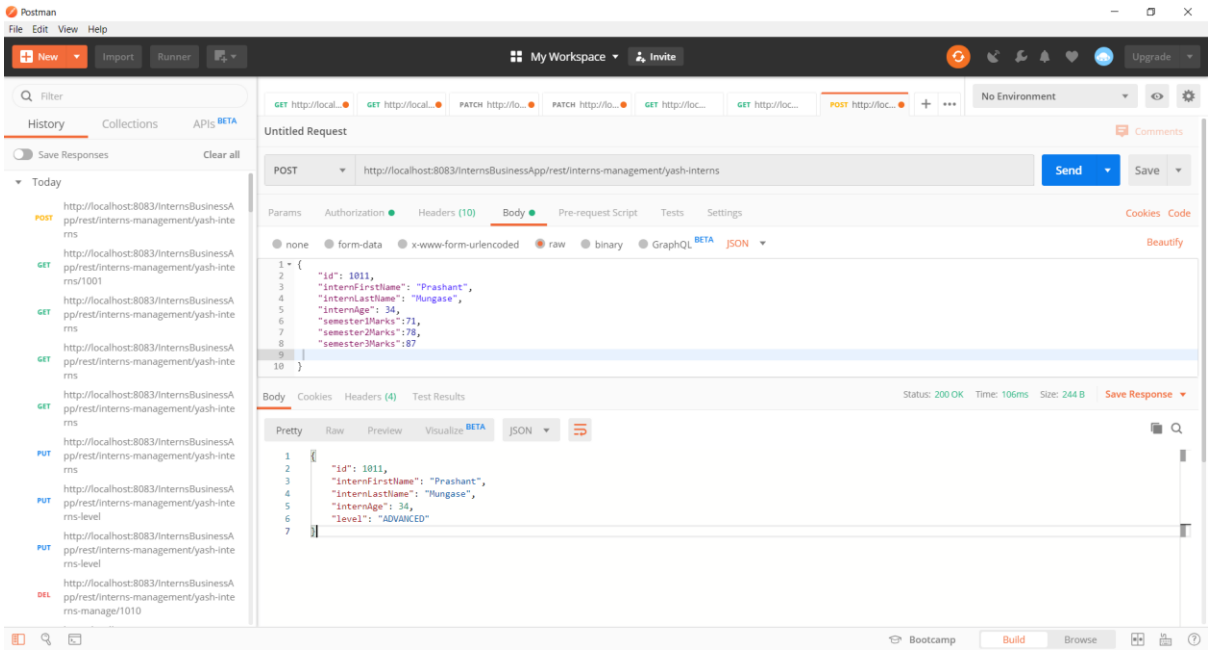

Postman
File Edit View Help
New Import Runner
My Workspace Invite
Upgrade
History Collections APIs BETA
Save Responses Clear all
Today
No Environment
GET http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns
Send Save
Pretty Raw Preview Visualize BETA JSON
{
"id": 1001,
"internFirstName": "Sabbir",
"internLastName": "Poonawala",
"internAge": 34,
"level": "ADVANCED"
},
{
"id": 1002,
"internFirstName": "Amitt",
"internLastName": "Desai",
"internAge": 22,
"level": "INTERMEDIATE"
},
{
"id": 1003,
"internFirstName": "Rohit",
"internLastName": "Patel",
"internAge": 22,
"level": "INTERMEDIATE"
},
{
"id": 1004,
"internFirstName": "Sachiin",
"internLastName": "Patil",
"internAge": 21,
"level": "ADVANCED"
},
{
"id": 1005,
"internFirstName": "Rohit",
Bootcamp Build Browse

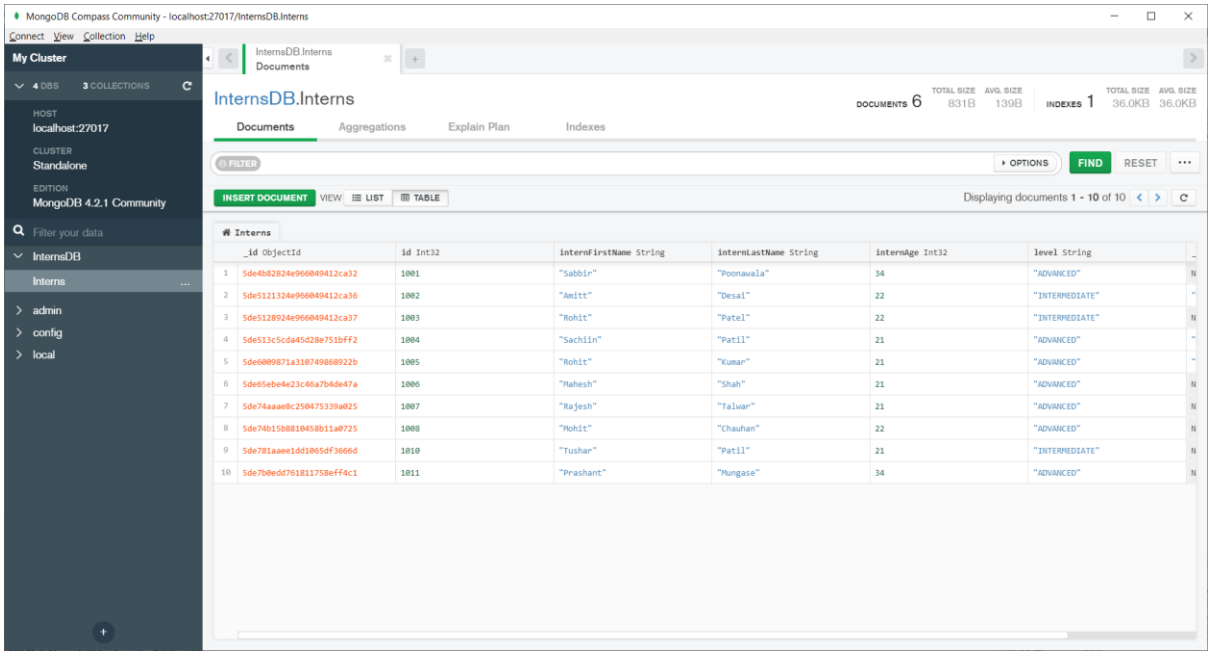

MongoDB Compass Community - localhost:27017/InternsDB.Interns
Connect View Collection Help
My Cluster
4 DBS 3 COLLECTIONS
HOST localhost:27017
CLUSTER Standalone
EDITION MongoDB 4.2.1 Community
Filter your data
InternsDB
Interns
admin
config
local
InternsDB.Interns
Documents
DOCUMENTS 6 TOTAL SIZE 831B AVG. SIZE 139B
INDEXES 1 TOTAL SIZE 36.0KB AVG. SIZE 36.0KB
Documents Aggregations Explain Plan Indexes
FILTER OPTIONS FIND RESET
INSERT DOCUMENT VIEW LIST TABLE
Displaying documents 1 - 9 of 9
Interns
_id ObjectId
id Int32
internFirstName String
internLastName String
internAge Int32
level String
1 5de4b82824e966049412ca32 1001 "Sabbir" "Poonawala" 34 "ADVANCED"
2 5de5121324e966049412ca36 1002 "Amitt" "Desai" 22 "INTERMEDIATE"
3 5de5128924e966049412ca37 1003 "Rohit" "Patel" 22 "INTERMEDIATE"
4 5de513c5cda45d28e751bff2 1004 "Sachiin" "Patil" 21 "ADVANCED"
5 5de6009871a310749868922b 1005 "Rohit" "Kumar" 21 "ADVANCED"
6 5de65ebe4e23c46a7b4de47a 1006 "Mahesh" "Shah" 21 "ADVANCED"
7 5de74aaae8c250475339a025 1007 "Rajesh" "Talwar" 21 "ADVANCED"
8 5de74b15b8810458b11a0725 1008 "Mohit" "Chauhan" 22 "ADVANCED"
9 5de781aaee1dd1065df3666d 1010 "Tushar" "Patil" 21 "INTERMEDIATE"

GET: http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns/1001

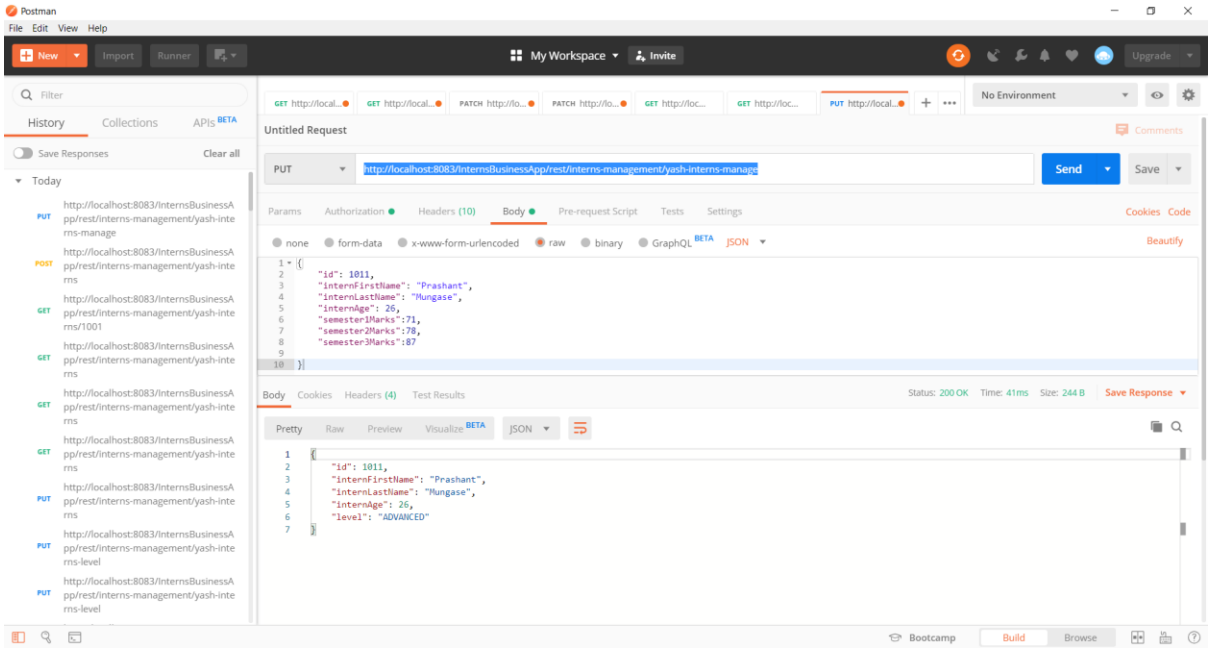


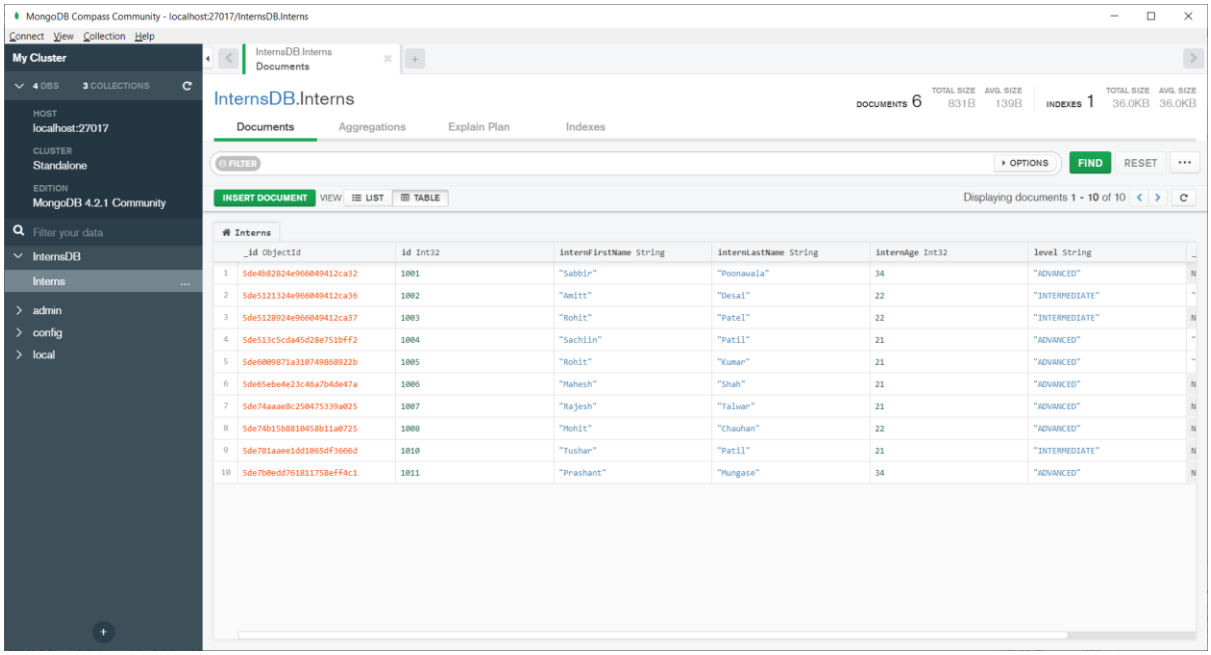
POST: http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns



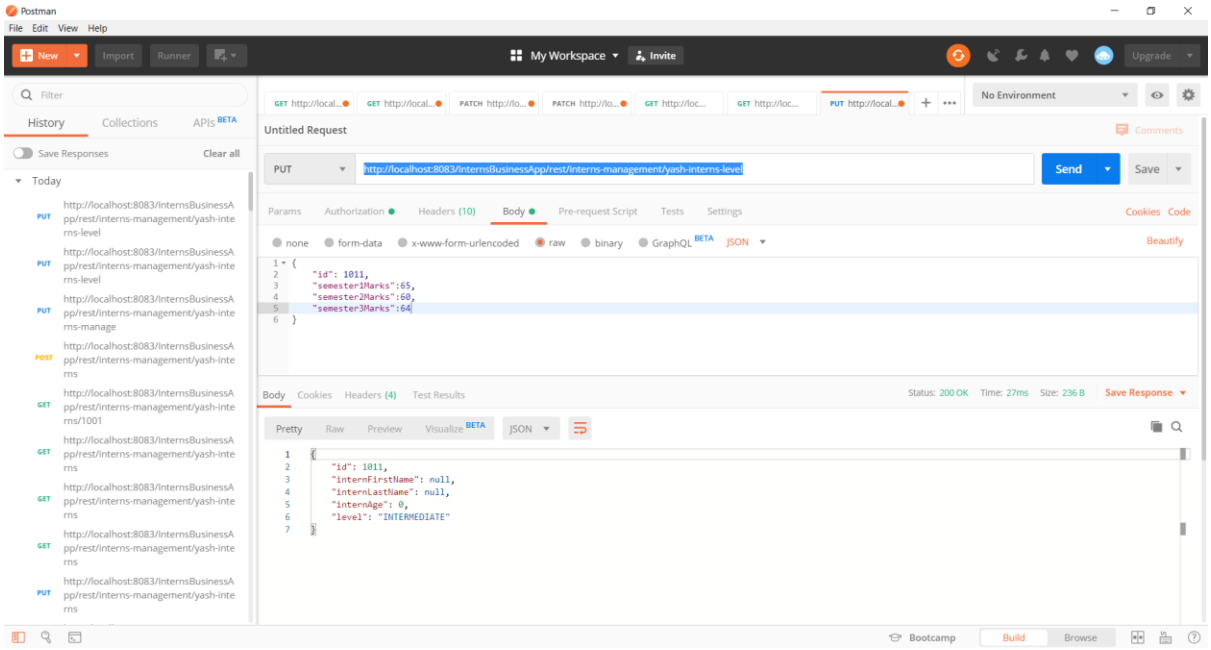


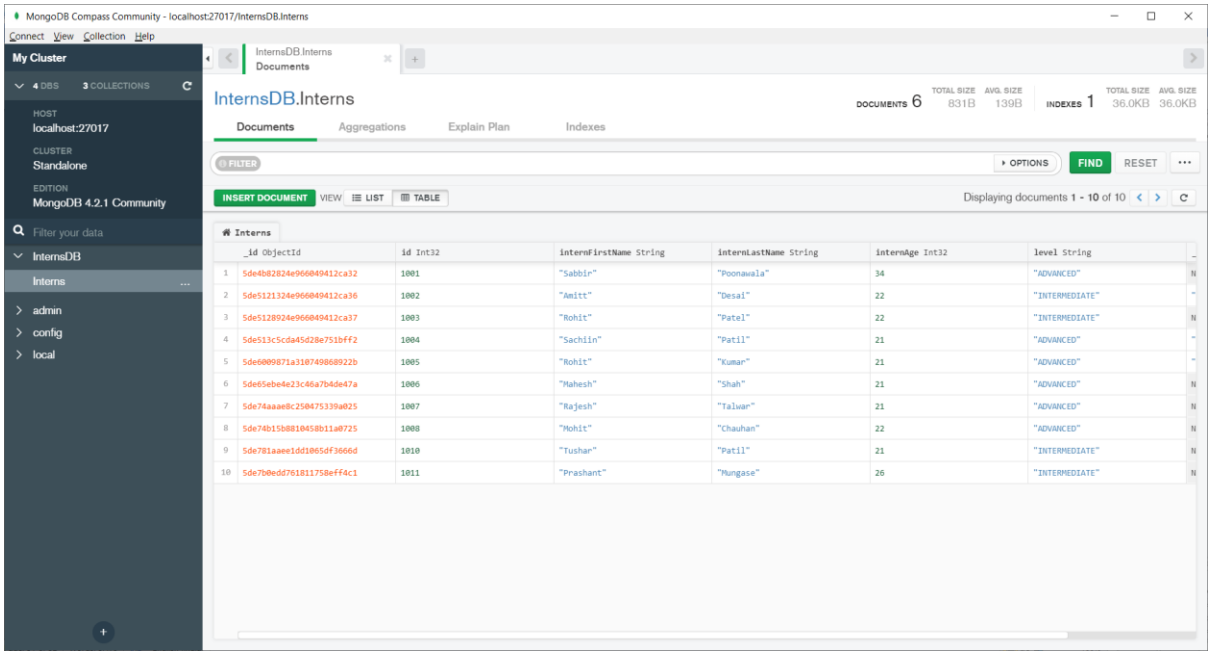
PUT: http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns-manage



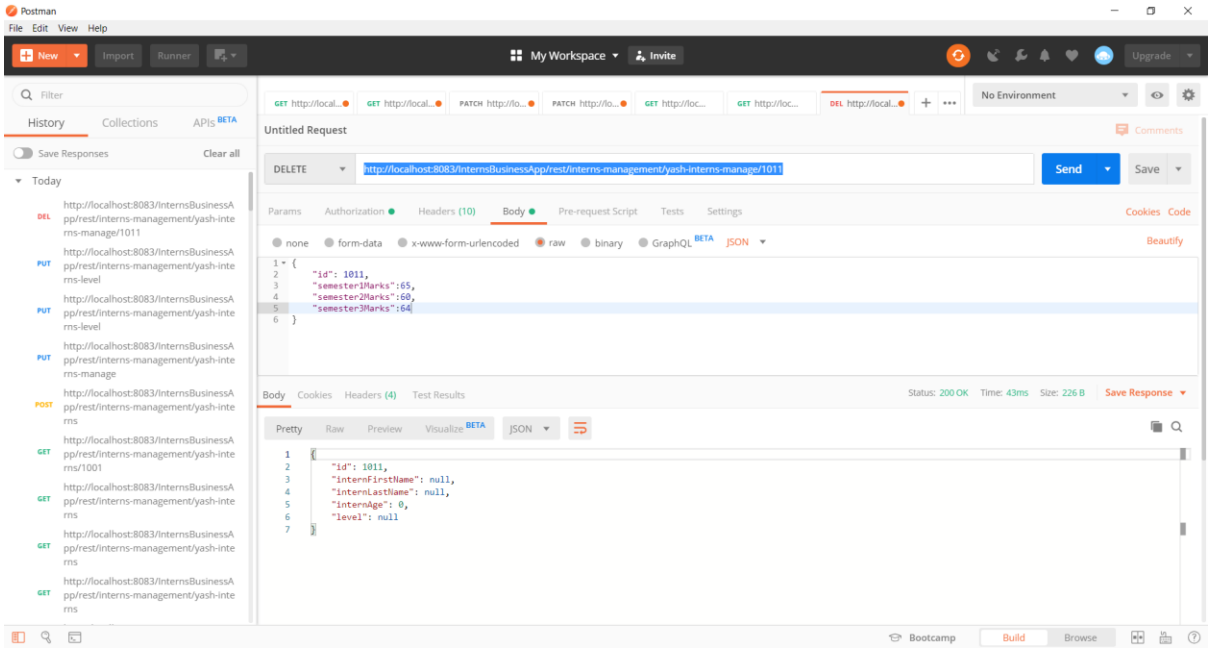


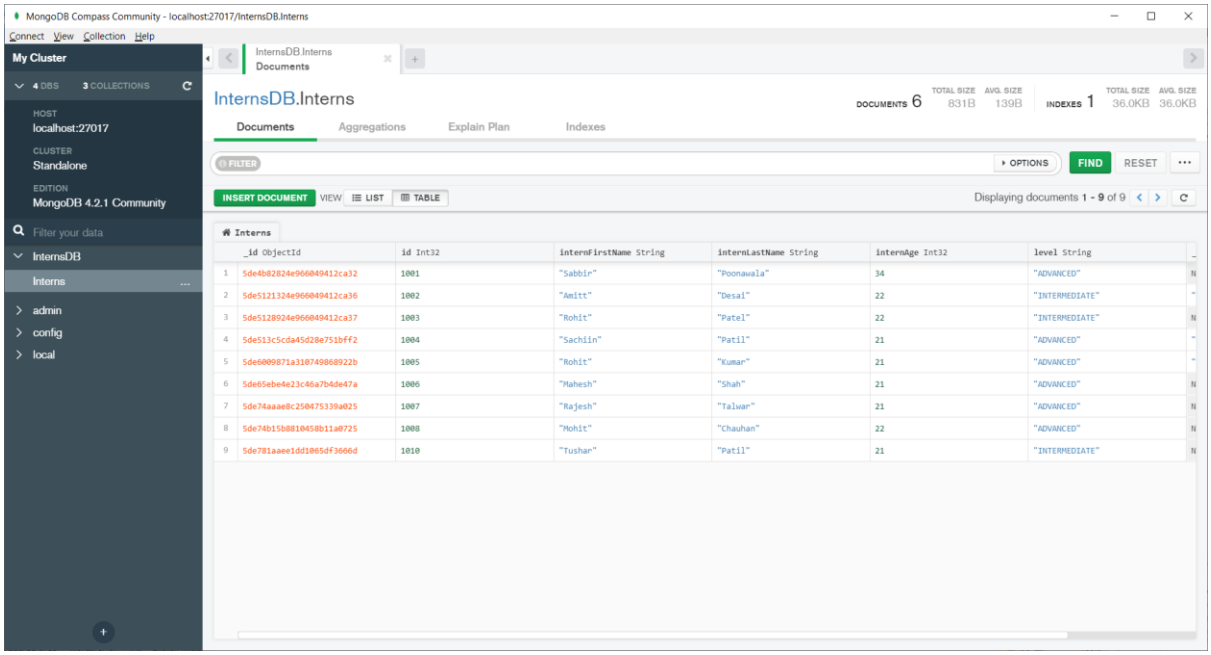
PUT: http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns-level





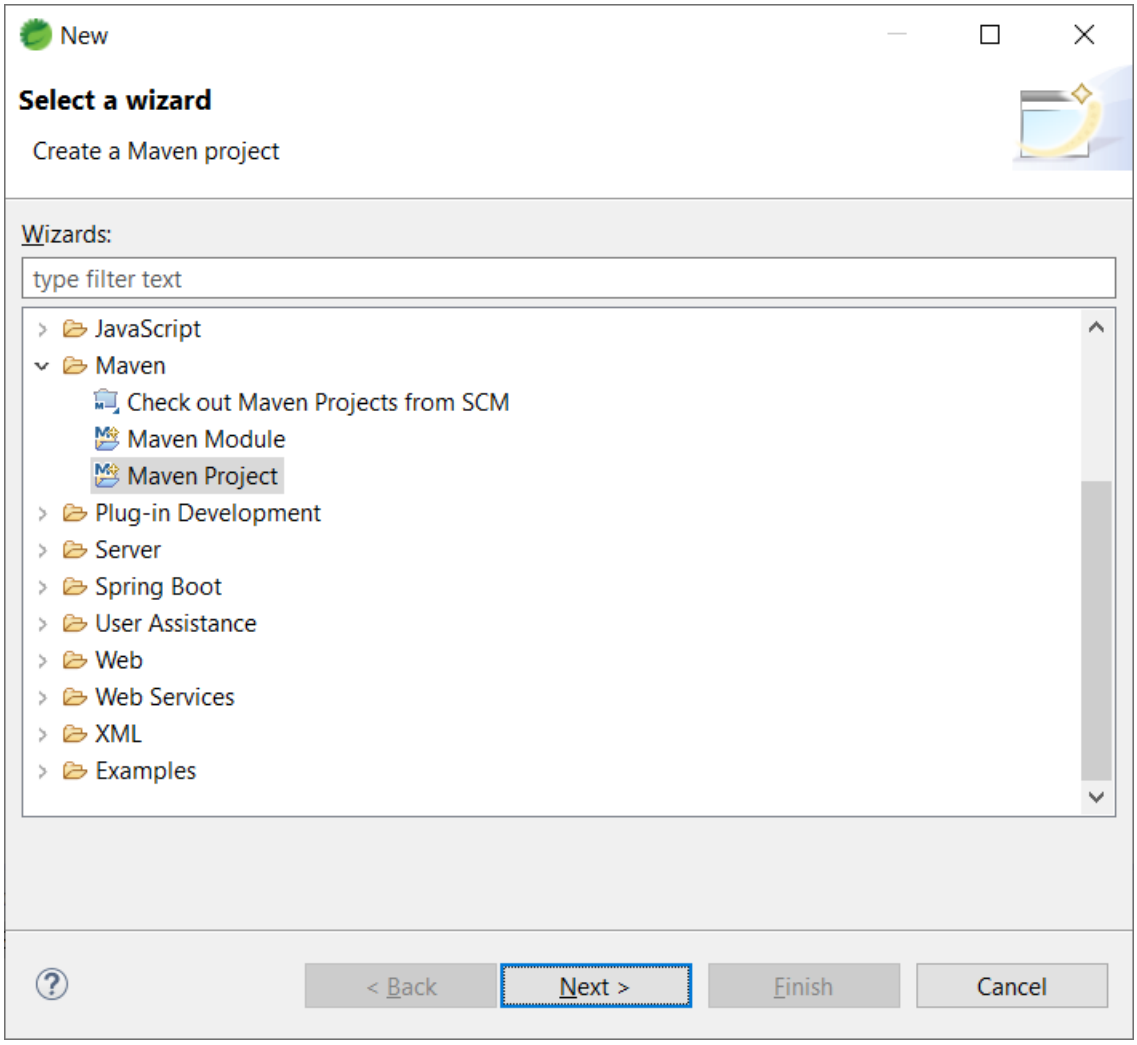
DELETE: http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns-manage/1011

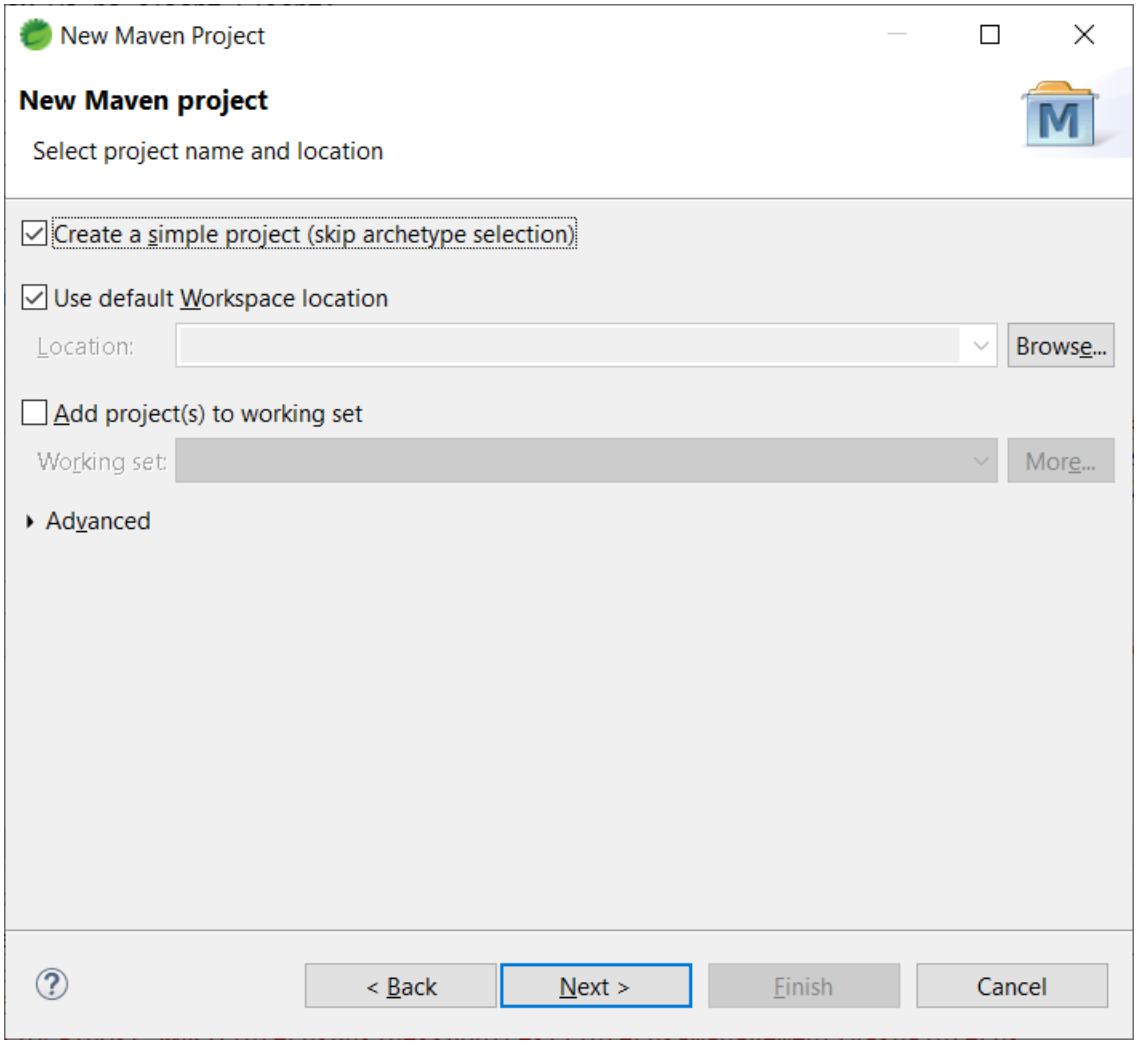


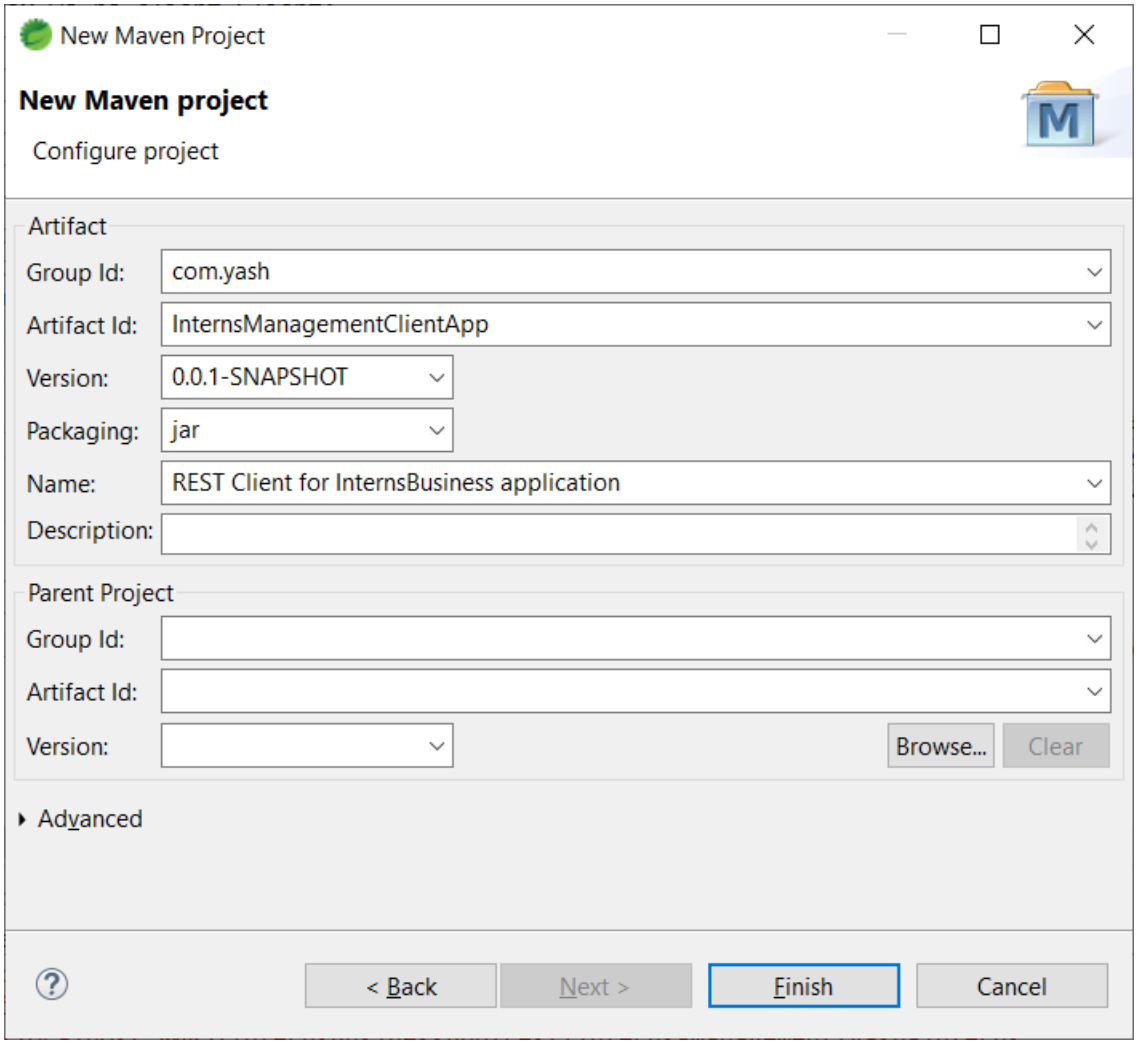


Rest Client using Jersey,

Create new maven project InternsManagementClient


New
Select a wizard
Create a Maven project
Wizards:
type filter text
JavaScript
Maven
Check out Maven Projects from SCM
Maven Module
Maven Project
Plug-in Development
Server
Spring Boot
User Assistance
Web
Web Services
XML
Examples
< Back
Next >
Finish
Cancel


New Maven Project
New Maven project
Select project name and location
Create a simple project (skip archetype selection)
Use default Workspace location
Location:
Browse...
Add project(s) to working set
Working set:
More...
Advanced
< Back
Next >
Finish
Cancel



Mention below dependencies in pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.yash</groupId>
  <artifactId>InternsManagementClient</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>InternsManagementClient</name>
  <description>Client application to access restful services</description>
  <dependencies>
  <dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.25.1</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.29.1</version>
</dependency>
```

```
</dependencies>
</project>
```

Copy com.yash.helper.Levels and com.yash.model from previous project to current project.

Note: We should ideally refrain from exposing business entity and its attributes information to client application. We will do work around in next section.

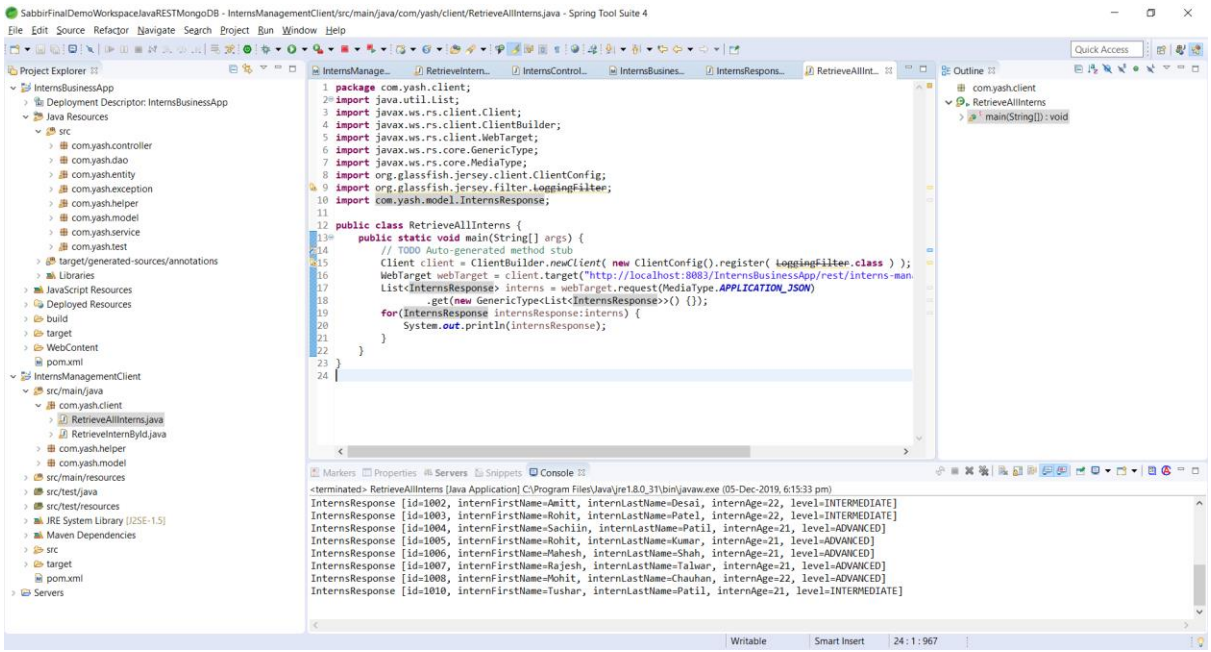
Create a package com.yash.client.

To retrieve all interns create a class `RetrieveAllInterns,`

```
package com.yash.client;
import java.util.List;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import org.glassfish.jersey.client.ClientConfig;
import org.glassfish.jersey.filter.LoggingFilter;
import com.yash.model.InternsResponse;

public class RetrieveAllInterns {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Client client = ClientBuilder.newClient( new
ClientConfig().register( LoggingFilter.class ) );
		WebTarget webTarget =
client.target("http://localhost:8083/InternsBusinessApp/rest/interns-
management").path("yash-interns");
		List<InternsResponse> interns =
webTarget.request(MediaType.APPLICATION_JSON)
				.get(new GenericType<List<InternsResponse>>() {});
		for(InternsResponse internsResponse:interns) {
			System.out.println(internsResponse);
		}
	}
}
```

Ensure tomcat server is running and InternBusinessApp is up.

Run above class and observe output on console.

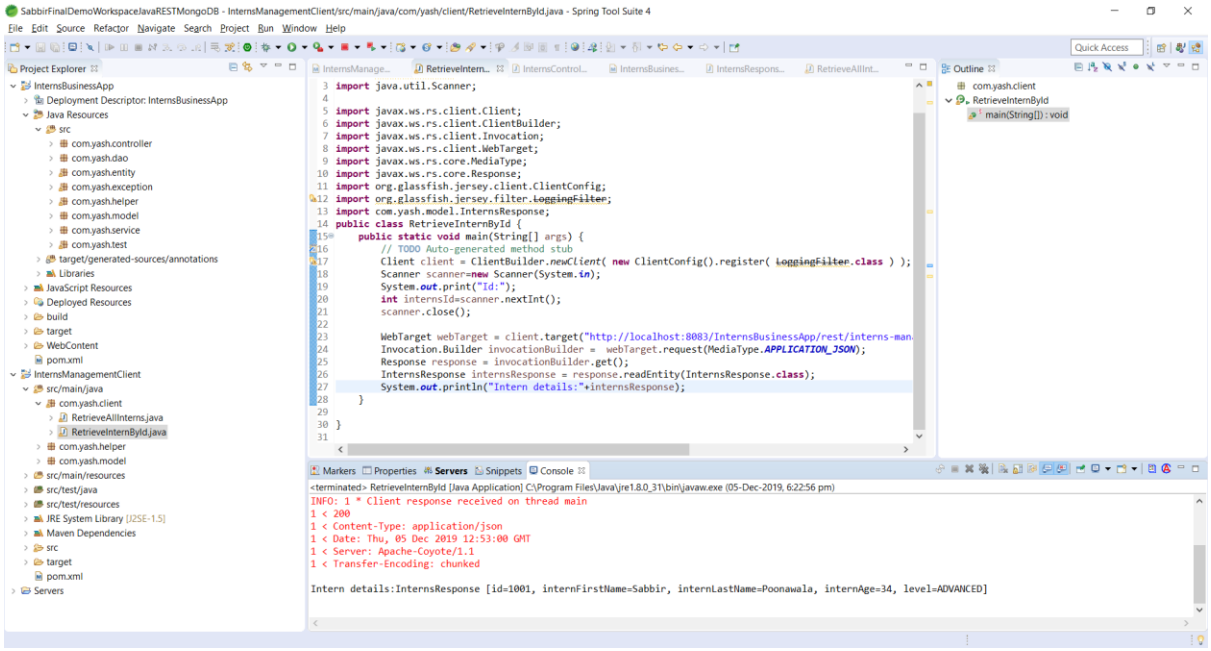


To retrieve intern details based on id, create a class RetrieveInternById

```
package com.yash.client;
import java.util.List;
import java.util.Scanner;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.glassfish.jersey.client.ClientConfig;
import org.glassfish.jersey.filter.LoggingFilter;
import com.yash.model.InternsResponse;
public class RetrieveInternById {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Client client = ClientBuilder.newClient( new
ClientConfig().register( LoggingFilter.class ) );
		Scanner scanner=new Scanner(System.in);
		System.out.print("Id:");
		int internsId=scanner.nextInt();
		scanner.close();

		WebTarget webTarget =
client.target("http://localhost:8083/InternsBusinessApp/rest/interns-
management").path("yash-interns").path(String.valueOf(internsId));
		Invocation.Builder invocationBuilder =
webTarget.request(MediaType.APPLICATION_JSON);
		Response response = invocationBuilder.get();
		InternsResponse internsResponse =
response.readEntity(InternsResponse.class);
```

```
            System.out.println("Intern details:"+internsResponse);
        }

}
```

Enter intern id as input and observe output

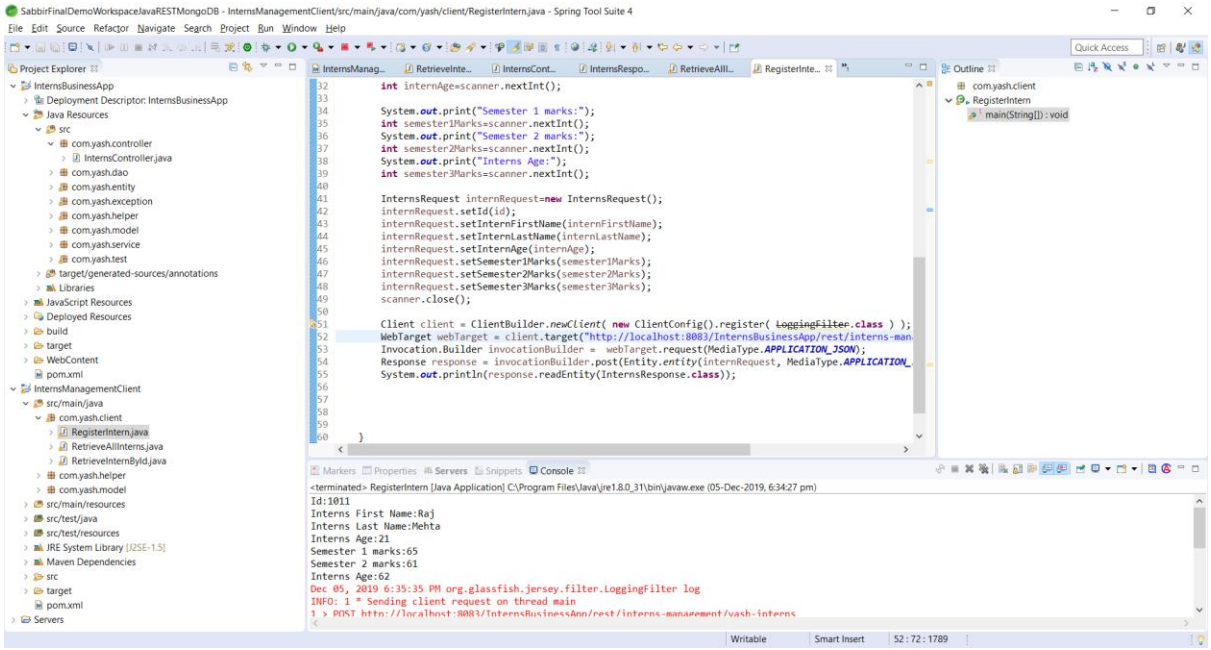


To register intern create a class RegisterIntern,

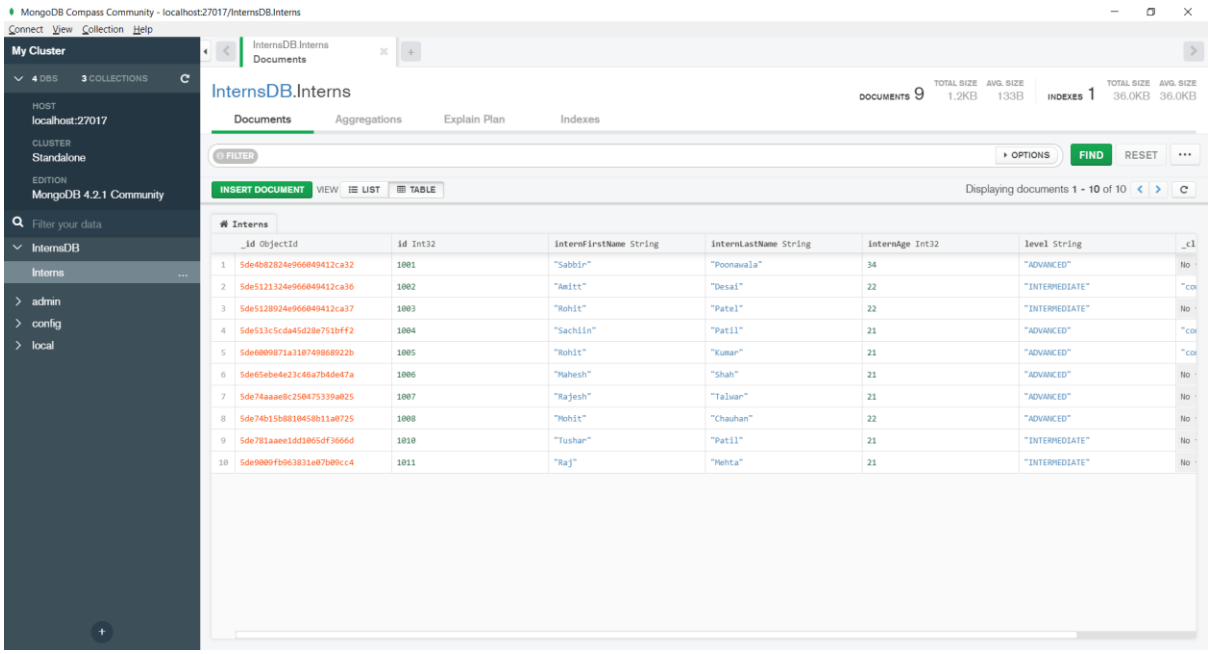
```
package com.yash.client;

import java.util.Scanner;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import org.glassfish.jersey.client.ClientConfig;
import org.glassfish.jersey.filter.LoggingFilter;

import com.yash.model.InternsRequest;
import com.yash.model.InternsResponse;

public class RegisterIntern {
```

```
	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Scanner scanner=new Scanner(System.in);
		System.out.print("Id:");
		int id=scanner.nextInt();
		System.out.print("Interns First Name:");
		String internFirstName=scanner.next();
		System.out.print("Interns Last Name:");
		String internLastName=scanner.next();
		System.out.print("Interns Age:");
		int internAge=scanner.nextInt();

		System.out.print("Semester 1 marks:");
		int semester1Marks=scanner.nextInt();
		System.out.print("Semester 2 marks:");
		int semester2Marks=scanner.nextInt();
		System.out.print("Interns Age:");
		int semester3Marks=scanner.nextInt();

		InternsRequest internRequest=new InternsRequest();
		internRequest.setId(id);
		internRequest.setInternFirstName(internFirstName);
		internRequest.setInternLastName(internLastName);
		internRequest.setInternAge(internAge);
		internRequest.setSemester1Marks(semester1Marks);
		internRequest.setSemester2Marks(semester2Marks);
		internRequest.setSemester3Marks(semester3Marks);
		scanner.close();

		Client client = ClientBuilder.newClient( new
ClientConfig().register( LoggingFilter.class ) );
		WebTarget webTarget =
client.target("http://localhost:8083/InternsBusinessApp/rest/interns-
management").path("yash-interns");
		Invocation.Builder invocationBuilder =
webTarget.request(MediaType.APPLICATION_JSON);
		Response response =
invocationBuilder.post(Entity.entity(internRequest, MediaType.APPLICATION_JSON));
		System.out.println(response.readEntity(InternsResponse.class));


	}

}
```

Run above class and give inputs on console,



Observe output that is intern details registered. Verify in MongoDB



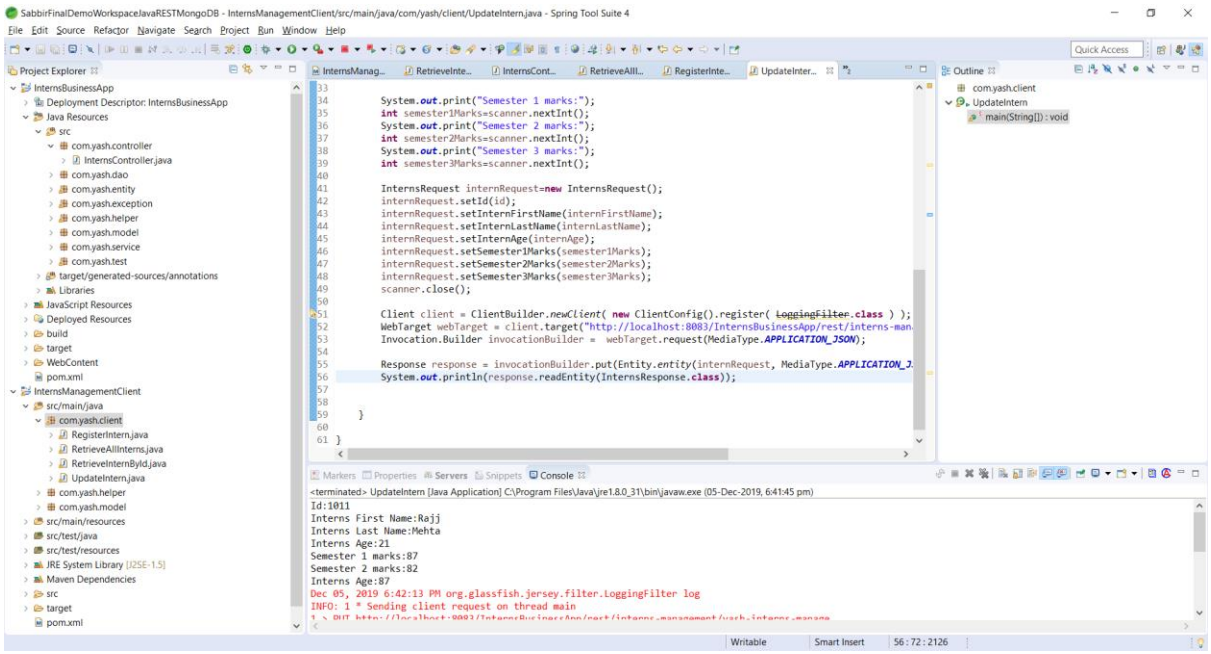
For updating intern details, create a class UpdateIntern,

```
package com.yash.client;

import java.util.Scanner;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
```

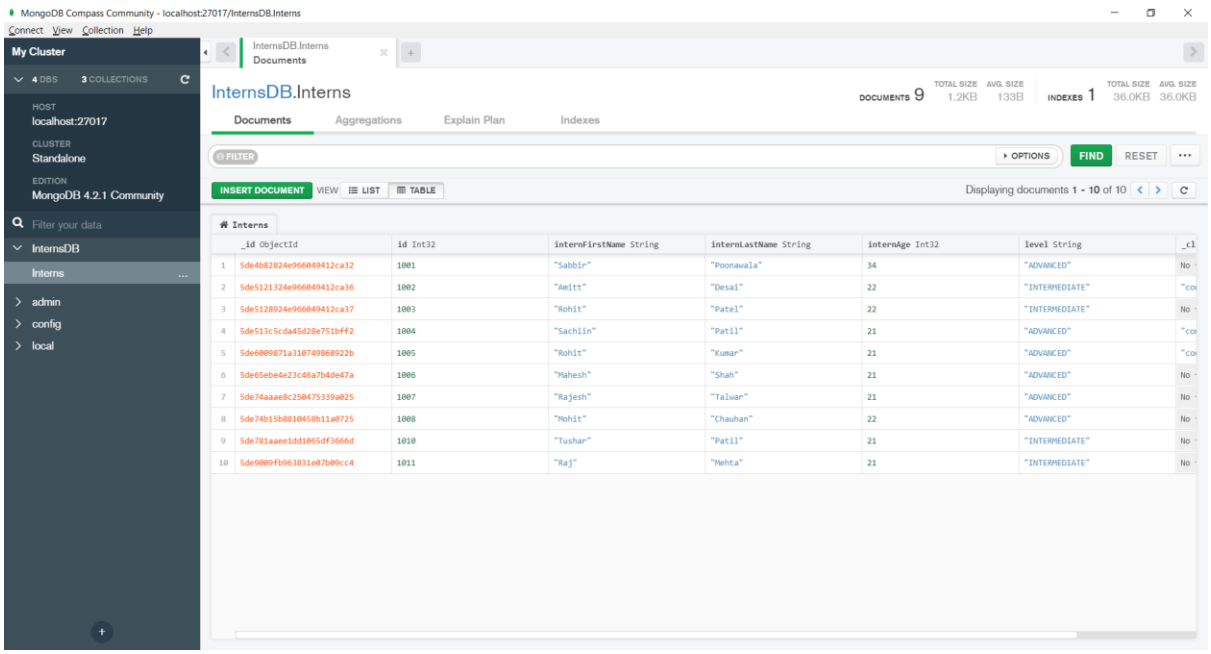
```
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import org.glassfish.jersey.client.ClientConfig;
import org.glassfish.jersey.filter.LoggingFilter;

import com.yash.model.InternsRequest;
import com.yash.model.InternsResponse;

public class UpdateIntern {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Scanner scanner=new Scanner(System.in);
		System.out.print("Id:");
		int id=scanner.nextInt();
		System.out.print("Interns First Name:");
		String internFirstName=scanner.next();
		System.out.print("Interns Last Name:");
		String internLastName=scanner.next();
		System.out.print("Interns Age:");
		int internAge=scanner.nextInt();

		System.out.print("Semester 1 marks:");
		int semester1Marks=scanner.nextInt();
		System.out.print("Semester 2 marks:");
		int semester2Marks=scanner.nextInt();
		System.out.print("Semester 3 marks:");
		int semester3Marks=scanner.nextInt();

		InternsRequest internRequest=new InternsRequest();
		internRequest.setId(id);
		internRequest.setInternFirstName(internFirstName);
		internRequest.setInternLastName(internLastName);
		internRequest.setInternAge(internAge);
		internRequest.setSemester1Marks(semester1Marks);
		internRequest.setSemester2Marks(semester2Marks);
		internRequest.setSemester3Marks(semester3Marks);
		scanner.close();

		Client client = ClientBuilder.newClient( new
ClientConfig().register( LoggingFilter.class ) );
		WebTarget webTarget =
client.target("http://localhost:8083/InternsBusinessApp/rest/interns-
management").path("yash-interns-manage");
		Invocation.Builder invocationBuilder =
webTarget.request(MediaType.APPLICATION_JSON);

		Response response =
invocationBuilder.put(Entity.entity(internRequest, MediaType.APPLICATION_JSON));
		System.out.println(response.readEntity(InternsResponse.class));

	}
```

```
}
```

Run above class and give input on console,



Verify in mongodb,

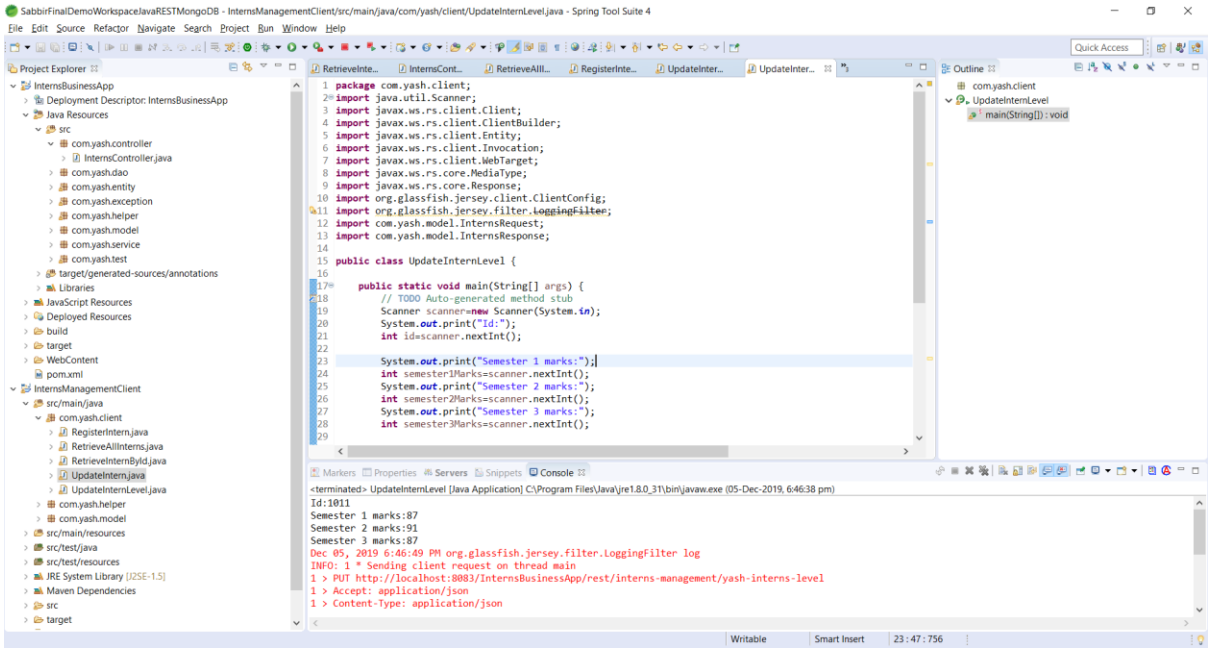


To update intern level, create a class UpdateInternLevel,

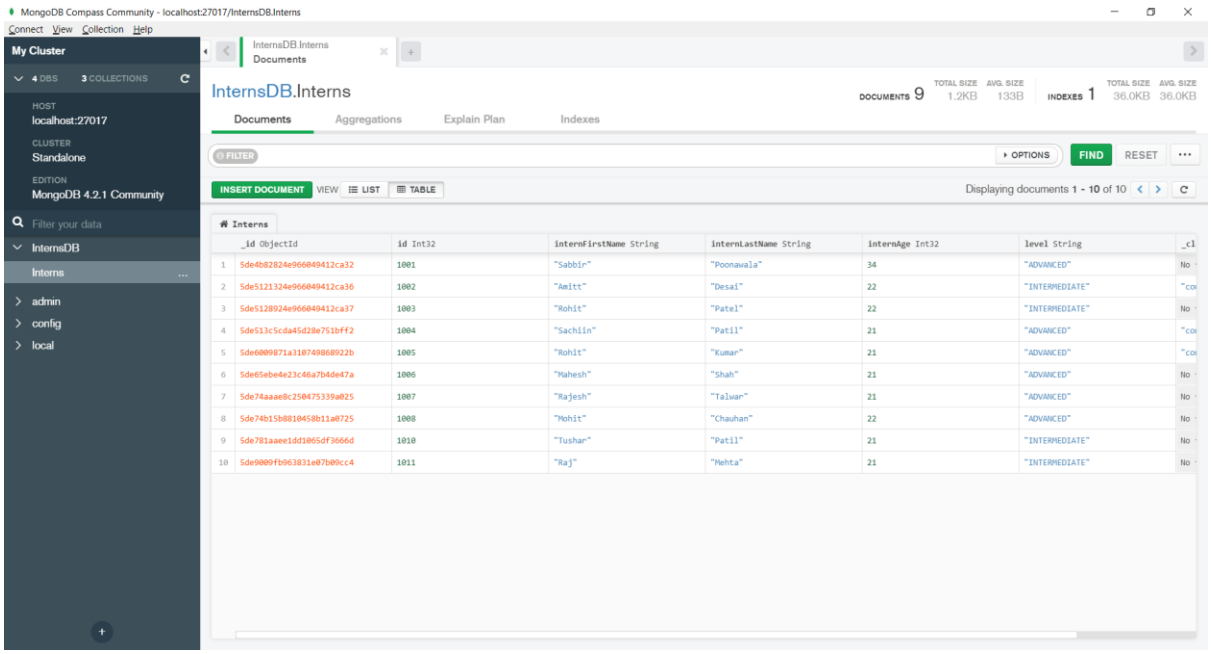
```
package com.yash.client;
import java.util.Scanner;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
```

```
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.glassfish.jersey.client.ClientConfig;
import org.glassfish.jersey.filter.LoggingFilter;
import com.yash.model.InternsRequest;
import com.yash.model.InternsResponse;

public class UpdateInternLevel {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Scanner scanner=new Scanner(System.in);
		System.out.print("Id:");
		int id=scanner.nextInt();

		System.out.print("Semester 1 marks:");
		int semester1Marks=scanner.nextInt();
		System.out.print("Semester 2 marks:");
		int semester2Marks=scanner.nextInt();
		System.out.print("Semester 3 marks:");
		int semester3Marks=scanner.nextInt();

		InternsRequest internRequest=new InternsRequest();
		internRequest.setId(id);
		internRequest.setSemester1Marks(semester1Marks);
		internRequest.setSemester2Marks(semester2Marks);
		internRequest.setSemester3Marks(semester3Marks);
		scanner.close();

		Client client = ClientBuilder.newClient( new
ClientConfig().register( LoggingFilter.class ) );
		WebTarget webTarget =
client.target("http://localhost:8083/InternsBusinessApp/rest/interns-
management").path("yash-interns-level");
		Invocation.Builder invocationBuilder =
webTarget.request(MediaType.APPLICATION_JSON);

		Response response =
invocationBuilder.put(Entity.entity(internRequest, MediaType.APPLICATION_JSON));
		System.out.println(response.readEntity(InternsResponse.class));

	}

}
```

Run above class and observe output on console,



Verify in mongodb,

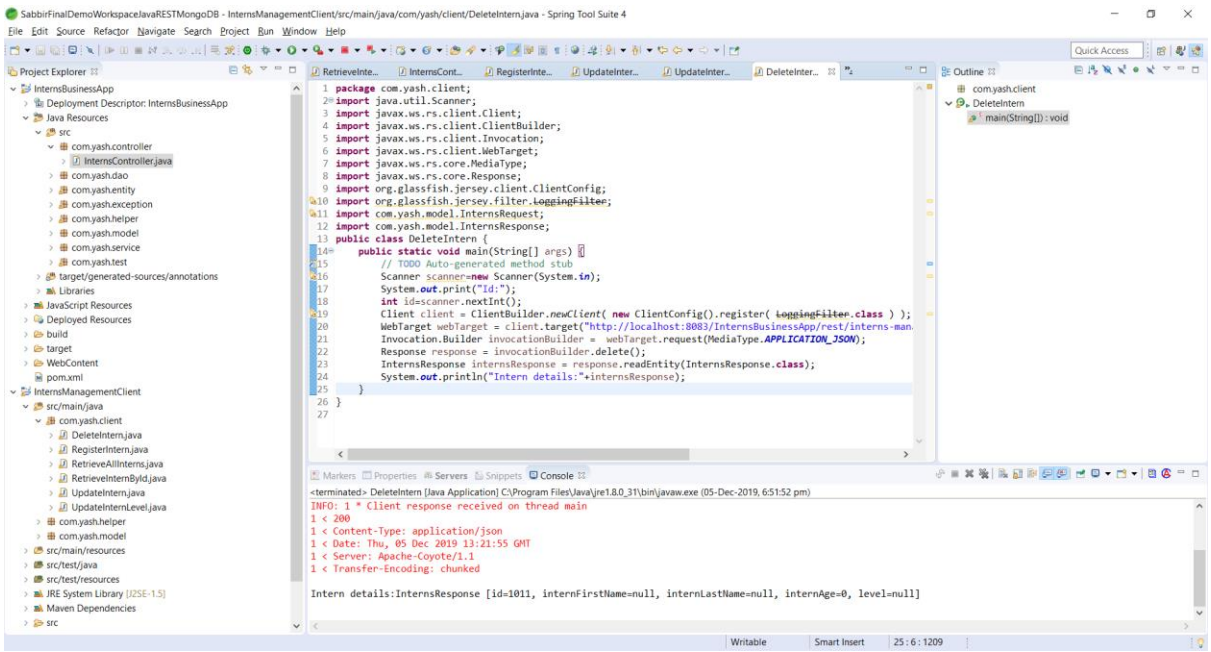


To delete intern details, create a class RemoveIntern,

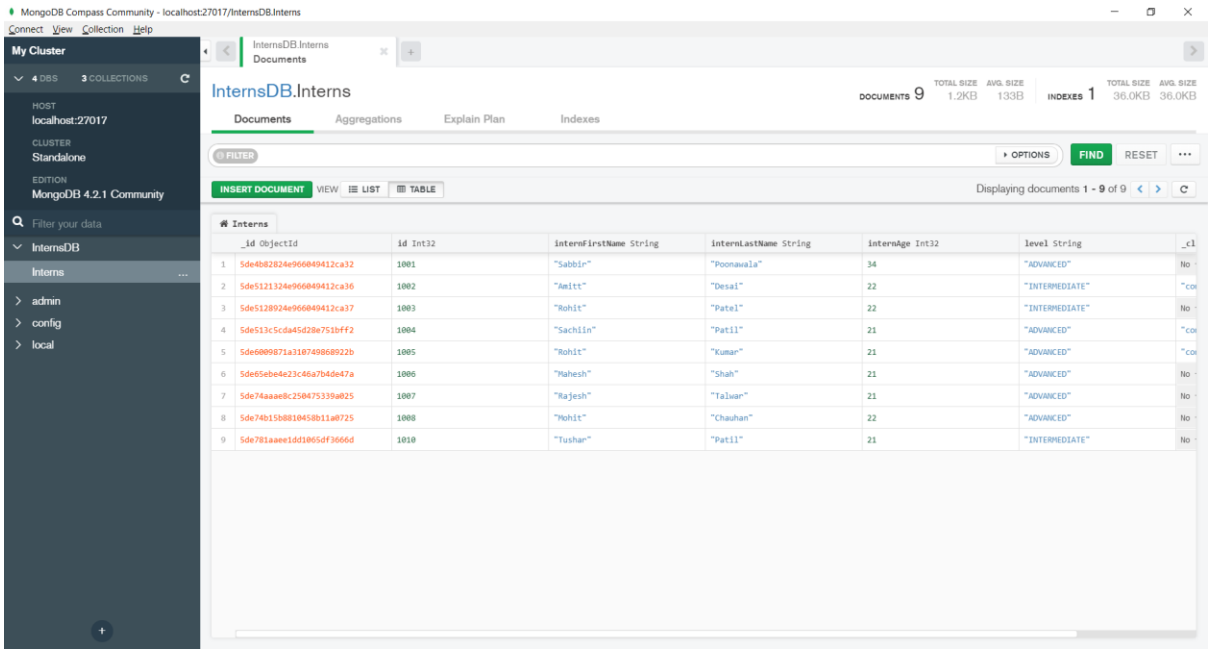
```
package com.yash.client;
import java.util.Scanner;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
```

```
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.glassfish.jersey.client.ClientConfig;
import org.glassfish.jersey.filter.LoggingFilter;
import com.yash.model.InternsRequest;
import com.yash.model.InternsResponse;
public class DeleteIntern {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Scanner scanner=new Scanner(System.in);
		System.out.print("Id:");
		int id=scanner.nextInt();
		Client client = ClientBuilder.newClient( new
ClientConfig().register( LoggingFilter.class ) );
		WebTarget webTarget =
client.target("http://localhost:8083/InternsBusinessApp/rest/interns-
management").path("yash-interns-manage").path(String.valueOf(id));
		Invocation.Builder invocationBuilder =
webTarget.request(MediaType.APPLICATION_JSON);
		Response response = invocationBuilder.delete();
		InternsResponse internsResponse =
response.readEntity(InternsResponse.class);
		System.out.println("Intern details:"+internsResponse);
	}
}
```

Run above class and give id as input.



Verify in mongodb,



We have tested basic functionality of RESTFul services. Restful services have to be secured. In next section we will discuss how to secure restful services we have created in above example.

REST Security Design Principles

The paper “The Protection of Information in Computer Systems” by Jerome Saltzer and Michael Schroeder, put forth eight design principles for securing information in computer systems, as described in the following sections:

1. **Least Privilege:** An entity should only have the required set of permissions to perform the actions for which they are authorized, and no more. Permissions can be added as needed and should be revoked when no longer in use.

2. **Fail-Safe Defaults:** A user’s default access level to any resource in the system should be “denied” unless they’ve been granted a “permit” explicitly.

3. **Economy of Mechanism:** The design should be as simple as possible. All the component interfaces and the interactions between them should be simple enough to understand.

4. **Complete Mediation:** A system should validate access rights to all its resources to ensure that they’re allowed and should not rely on the cached permission matrix. If the access level to a given resource is being revoked, but that isn’t reflected in the permission matrix, it would violate the security.

5. **Open Design:** This principle highlights the importance of building a system in an open manner—with no secret, confidential algorithms.

6. **Separation of Privilege:** Granting permissions to an entity should not be purely based on a single condition, a combination of conditions based on the type of resource is a better idea.

7. **Least Common Mechanism:** It concerns the risk of sharing state among different components. If one can corrupt the shared state, it can then corrupt all the other components that depend on it.

8. **Psychological Acceptability:** It states that security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present. In short, security should not make worse the user experience.

Best Practices to Secure REST APIs

Below given points may serve as a checklist for designing the security mechanism for REST APIs.

Keep it Simple

Secure an API/System – just how secure it needs to be. Every time you make the solution more complex "unnecessarily," you are also likely to leave a hole.

Always Use HTTPS

By always using SSL, the authentication credentials can be simplified to a randomly generated access token that is delivered in the username field of HTTP Basic Auth. It's relatively simple to use, and you get a lot of security features for free.

If you use HTTP 2, to improve performance – you can even send multiple requests over a single connection, that way you avoid the complete TCP and SSL handshake overhead on later requests.

Use Password Hash

Passwords must always be hashed to protect the system (or minimize the damage) even if it is compromised in some hacking attempts. There are many such hashing algorithms which can prove really effective for password security e.g. PBKDF2, bcrypt and scrypt algorithms.

Never expose information on URLs

Usernames, passwords, session tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them easily exploitable.

```
https://api.domain.com/user-management/users/{id}/someAction?apiKey=abcd123456789
//Very BAD !!
```

The above URL exposes the API key. So, never use this form of security.

Consider OAuth

Though basic auth is good enough for most of the APIs and if implemented correctly, it's secure as well – yet you may want to consider OAuth as well. The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

Consider Adding Timestamp in Request

Along with other request parameters, you may add a request timestamp as an HTTP custom header in API requests. The server will compare the current timestamp to the request timestamp and only accepts the request if it is within a reasonable timeframe (1-2 minutes, perhaps).

This will prevent very basic replay attacks from people who are trying to brute force your system without changing this timestamp.

Input Parameter Validation

Validate request parameters on the very first step, before it reaches to application logic. Put strong validation checks and reject the request immediately if validation fails. In API response, send relevant error messages and example of correct input format to improve user experience.

Eclipse will publish the applications to the workspace folder. Ideally we should not modify configuration of tomcat in program files folder in development environment.

For example, the configuration of tomcat is copied from

C:\Program Files\Apache Software Foundation\Tomcat 8.0\conf

to

D:\Sabbir\SabbirFinalDemoWorkspaceJavaRESTMongoDB\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\conf

We have to configure Tomcat to support *container managed security*, by connecting to an existing "database" of usernames, passwords, and user roles.

A **Realm** is a "database" of usernames and passwords that identify valid users of a web application (or set of web applications), plus an enumeration of the list of *roles* associated with each valid user. You can think of roles as similar to *groups* in Unix-like operating systems, because access to specific web application resources is granted to all users possessing a particular role (rather than enumerating the list of associated usernames). A particular user can have any number of roles associated with their username.

The default configuration defines a Realm (UserDatabaseRealm) for the Catalina Engine, to perform user authentication for accessing this engine. It uses the JNDI name UserDatabase

defined in the GlobalNamingResources.
Besides the UserDatabaseRealm, there are: JDBCRealm (for authenticating users to connect to a relational database via the JDBC driver); DataSourceRealm (to connect to a DataSource via JNDI; JNDIRealm (to connect to an LDAP directory); and MemoryRealm (to load an XML file in memory).

We will use default configuration.

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
resourceName="UserDatabase"/>
</Realm>
```

To specify user, password and roles,

Open

<<WORKSPACE

FOLDER>>\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\conf\tomcat-users.xml

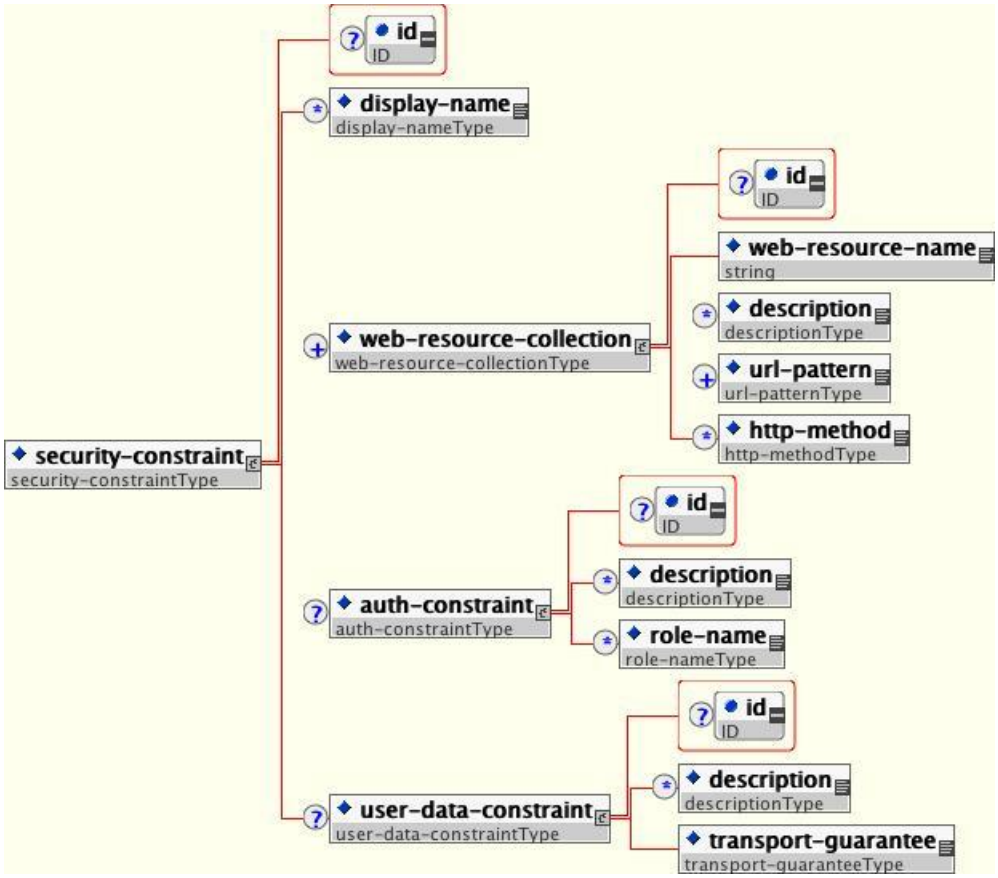
`<tomcat-users>`**:** This is the root element. This has two nested elements: role and user.
<role>: Each role that a user can play is defined with a <role> element. The attribute rolename specifies the name.
`<user>`**:** Each user has a <user> entry. This has three required attributes: username, password and roles. Note that a user can have more than one role.

- username – Username this user must log on with.
- password – Password this user must log on with (in clear text).
- roles – Comma-delimited list of the role names associated with this user.

Specify below usernames and roles.

```
<role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="JuniorRecruiter"/>
  <role rolename="SeniorRecruiter"/>
  <user username="tomcat" password="tomcat" roles="tomcat,manager-
gui,manager-status"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
  <user username="recruiter" password="yash123"
roles="JuniorRecruiter" />
  <user username="adminrecruiter" password="yash1234"
roles="SeniorRecruiter" />
```

In a web application, security is defined by the roles that are allowed access to content by a URL pattern that identifies the protected content. This set of information is declared by using the `web.xml security-constraint` element.



The content to be secured is declared using one or more web-resource-collection elements. Each web-resource-collection element contains an optional series of url-pattern elements followed by an optional series of http-method elements. The url-pattern element value specifies a URL pattern against which a request URL must match for the request to correspond to an attempt to access secured content. The http-method element value specifies a type of HTTP request to allow.

The optional user-data-constraint element specifies the requirements for the transport layer of the client to server connection. The requirement may be for content integrity (preventing data tampering in the communication process) or for confidentiality (preventing reading while in transit). The transport-guarantee element value specifies the degree to which communication between the client and server should be protected. Its values are NONE, INTEGRAL, and CONFIDENTIAL. A value of NONE means that the application does not require any transport guarantees. A value of INTEGRAL means that the application requires the data sent between the client and server to be sent in such a way that it can't be changed in transit. A value of CONFIDENTIAL means that the application requires the data to be transmitted in a fashion that prevents other entities from observing the contents of the transmission. In most cases, the presence of the INTEGRAL or CONFIDENTIAL flag indicates that the use of SSL is required.

Let’s apply security constraint on our service URI’s.

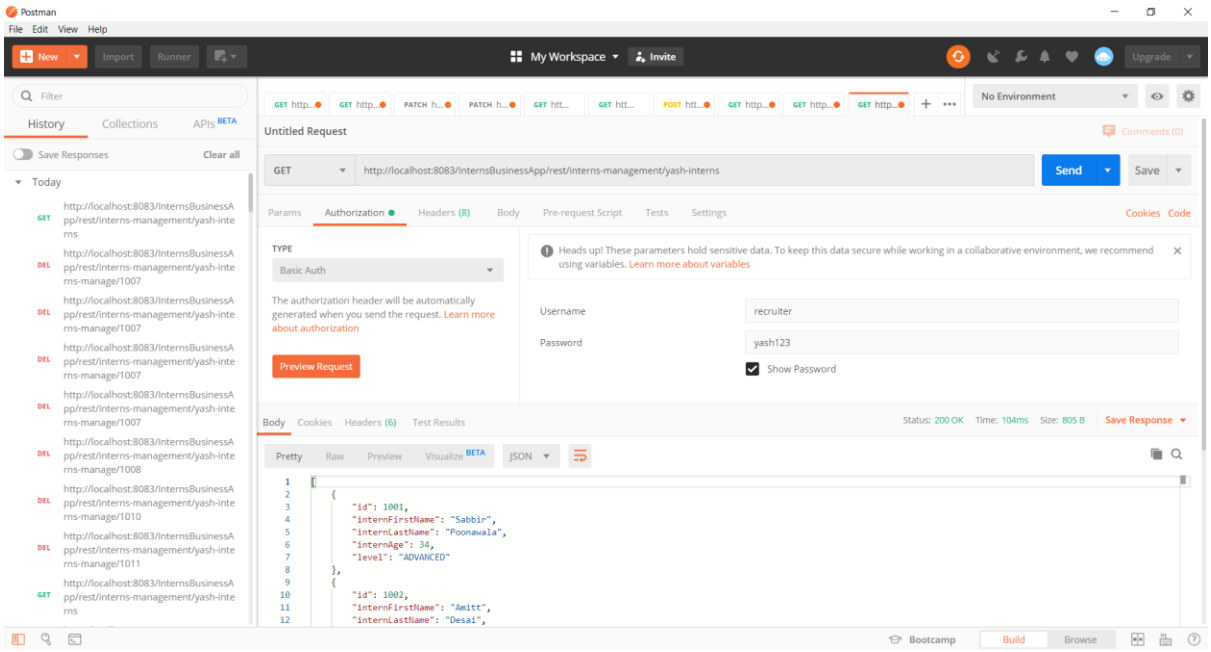
Retrieval of Interns data can be done by user with role “JuniorRecruiter” or “SeniorRecruiter”, but modifications can be done only by user with role “SeniorRecruiter”

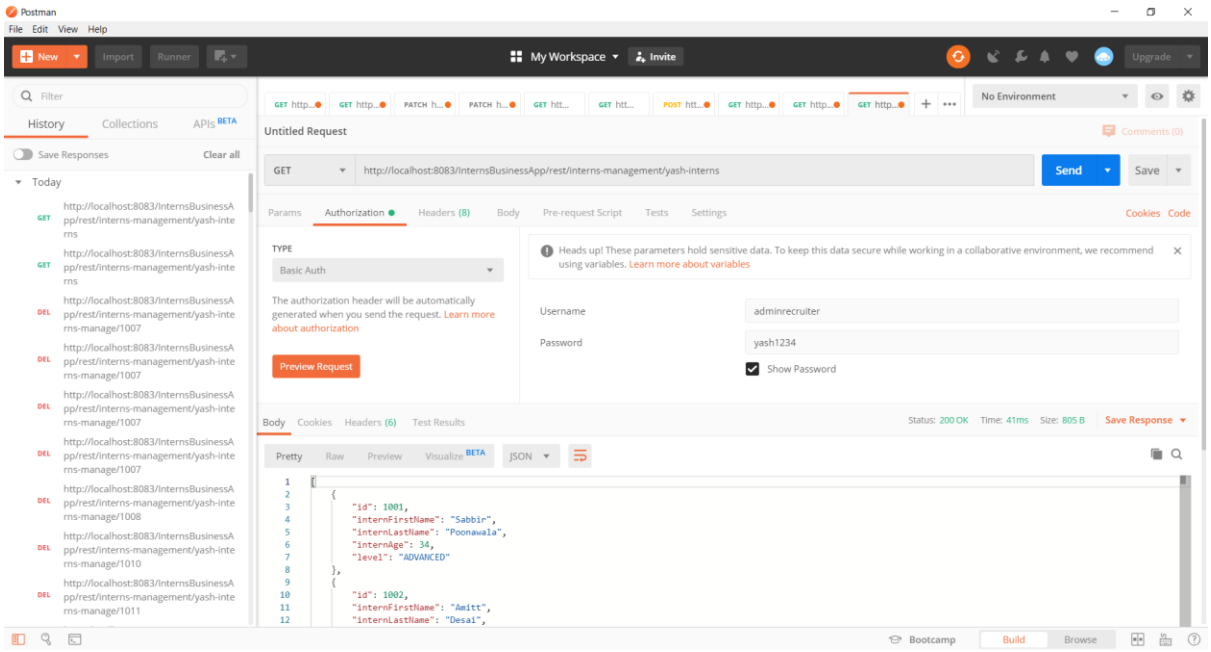
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID" version="3.1">
  <display-name>InternsBusinessApp</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <!-- Jersey Servlet configurations -->
      <servlet>
            <servlet-name>Jersey REST Service</servlet-name>
            <servlet-
class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
            <init-param>
                  <param-name>com.sun.jersey.config.property.packages</param-
name>
                  <param-value>com.yash.controller</param-value>
            </init-param>
            <init-param>
            <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
            <param-value>true</param-value>
      </init-param>
            <load-on-startup>1</load-on-startup>
      </servlet>
      <servlet-mapping>
            <servlet-name>Jersey REST Service</servlet-name>
            <url-pattern>/rest/*</url-pattern>
      </servlet-mapping>
      <!-- Jersey Servlet configurations -->

      <!-- This is where security features are being enabled -->
  <security-constraint>
    <display-name>Restricted GET Request</display-name>
    <web-resource-collection>
       <web-resource-name>Restricted GET request to Interns data</web-resource-
name>
       <url-pattern>/rest/interns-management/yash-interns</url-pattern>
       <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
       <role-name>JuniorRecruiter</role-name>
        <role-name>SeniorRecruiter</role-name>
    </auth-constraint>
    <user-data-constraint>
<!-- In production environment it is advised to set the guarantee as CONFIDENTIAL
-->
       <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
  <security-constraint>
    <display-name>Restricted POST Request</display-name>
    <web-resource-collection>
```

```
        <web-resource-name>Restricted POST request to Interns data</web-resource-
name>
        <url-pattern>/rest/interns-management/yash-interns</url-pattern>
        <http-method>POST</http-method>
    </web-resource-collection>
    <web-resource-collection>
    <web-resource-name>Restricted PUT request to Interns data</web-resource-name>
    <url-pattern>/rest/interns-management/yash-interns-manage</url-pattern>
    <http-method>PUT</http-method>
    </web-resource-collection>
     <web-resource-collection>
        <web-resource-name>Restricted PUT request to Interns data</web-resource-
name>
        <url-pattern>/rest/interns-management/yash-interns-level</url-pattern>
        <http-method>PUT</http-method>
    </web-resource-collection>
    <web-resource-collection>
        <web-resource-name>Restricted DELETE request of Interns data</web-
resource-name>
        <url-pattern>/rest/interns-management/yash-interns-manage/*</url-pattern>
        <http-method>DELETE</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>SeniorRecruiter</role-name>
    </auth-constraint>
    <user-data-constraint>
<!-- In production environment it is advised to set the guarantee as CONFIDENTIAL
-->
        <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
<!-- Using Basic authentication -->
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
 <security-role>
        <description>Normal operator recruiter</description>
        <role-name>JuniorRecruiter</role-name>
    </security-role>
      <security-role>
        <description>Admin recruiter</description>
        <role-name>SeniorRecruiter</role-name>
    </security-role>

</web-app>
```

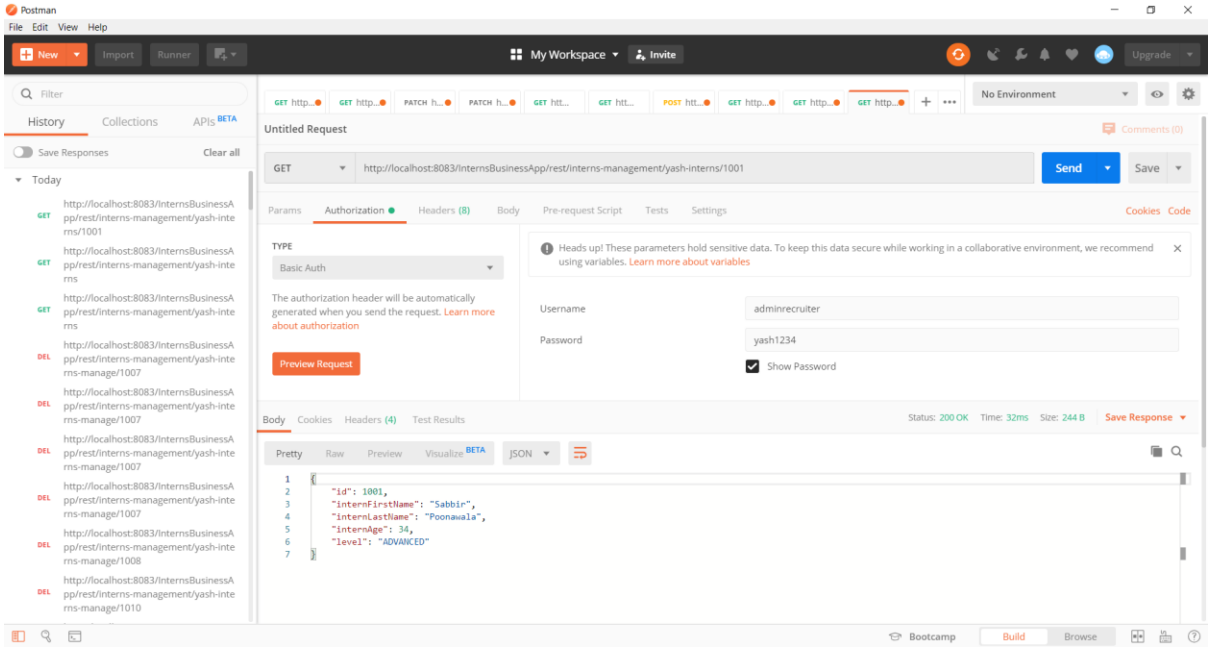
Test using Postman,

GET: http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns





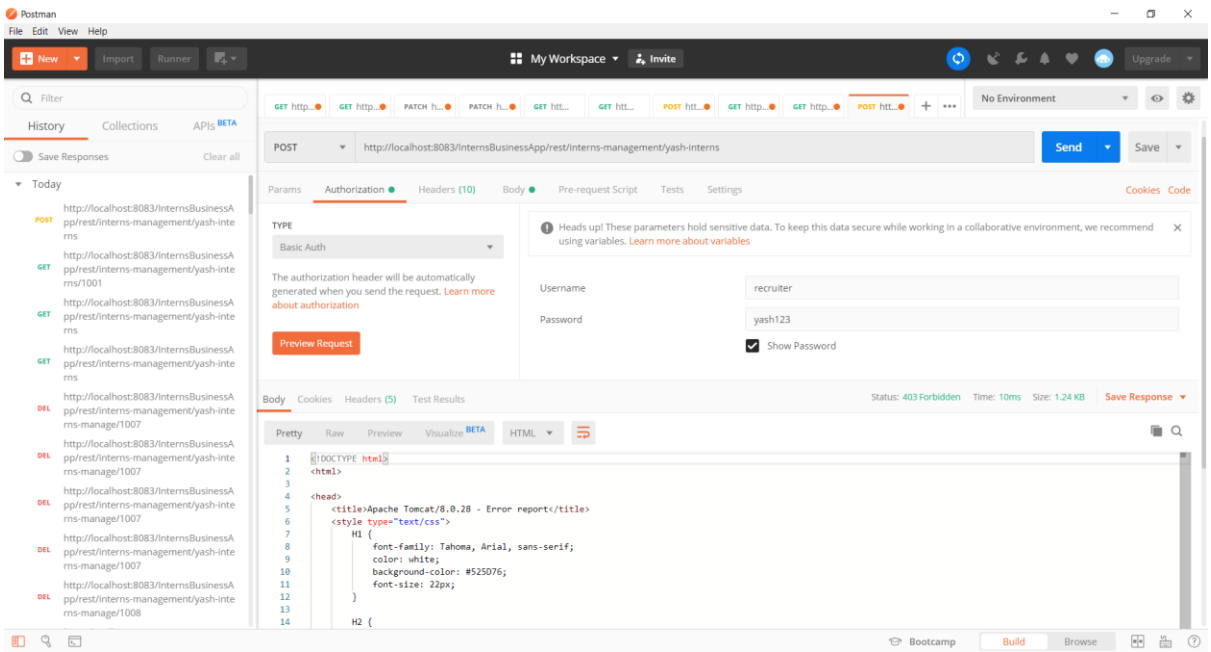
GET: http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns/1001



POST: http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns

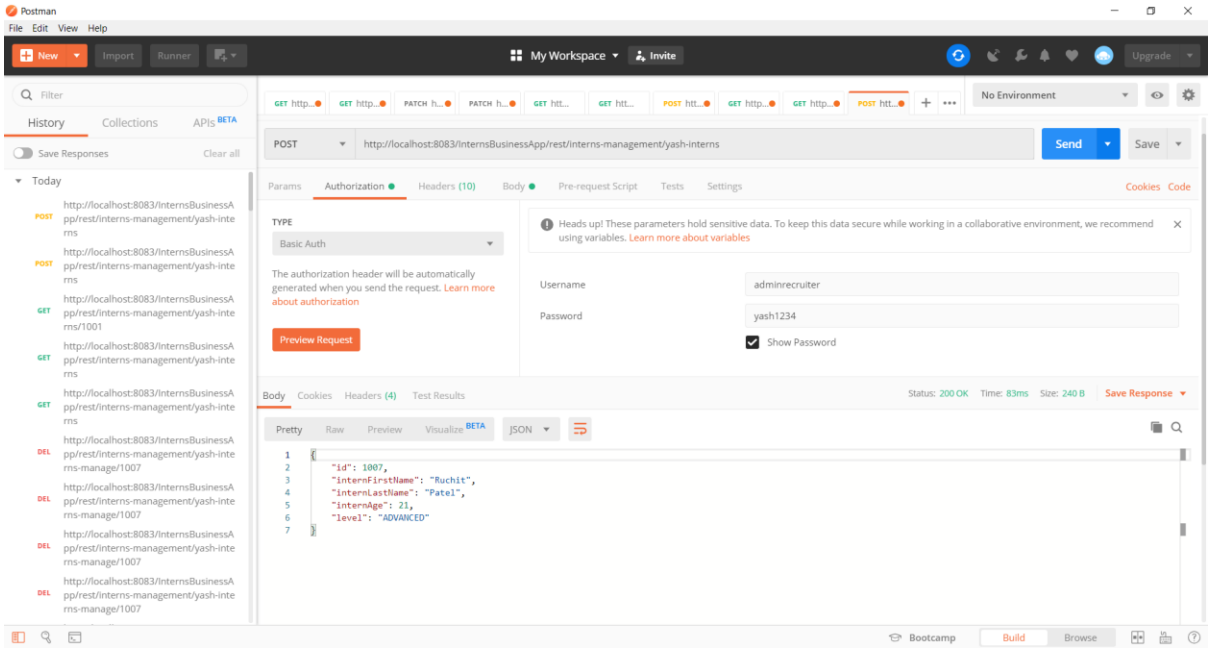
Test with user “recruiter” and password “yash123” who has a role “JuniorRecruiter”

HTTP ERROR STATUS IS 403 FORBIDDEN

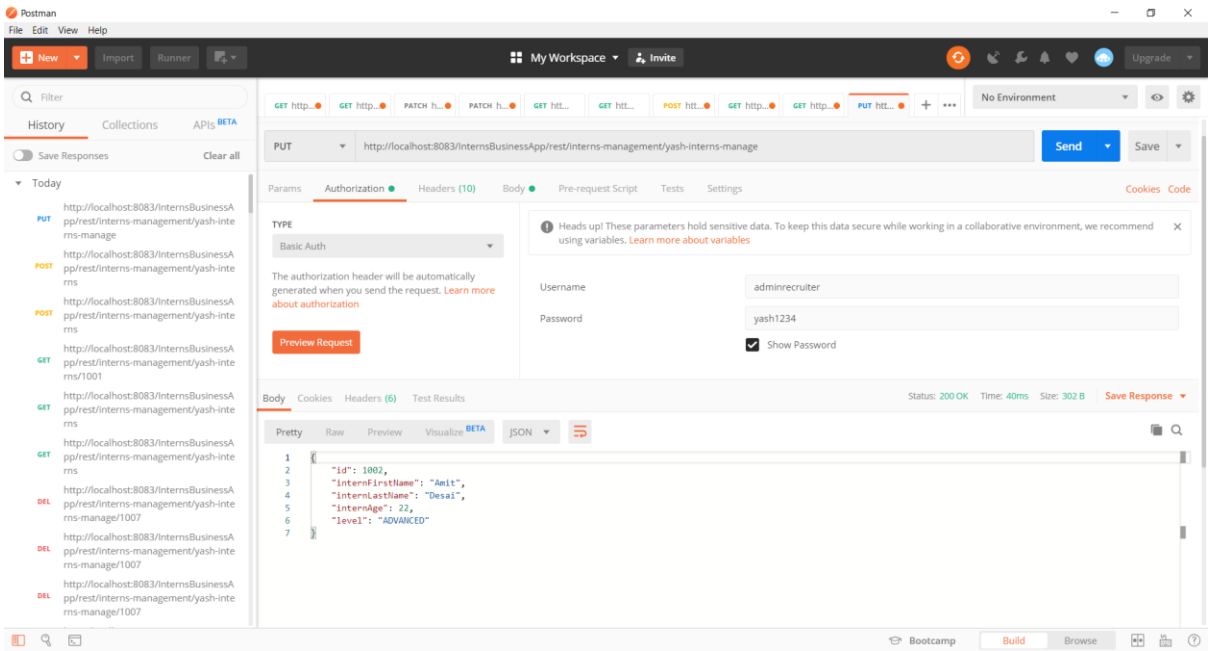


Test with user “adminrecruiter” and password “yash1234” who has a role “SeniorRecruiter”

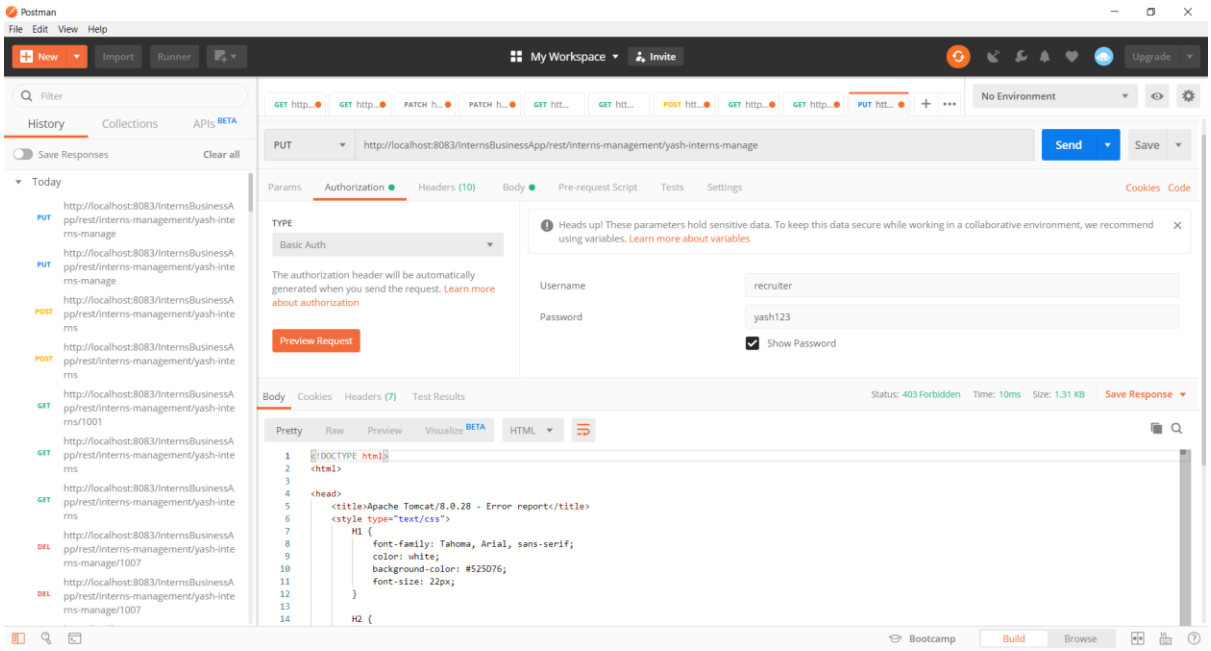
HTTP STATUS IS 200 OK



PUT: http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns-manage

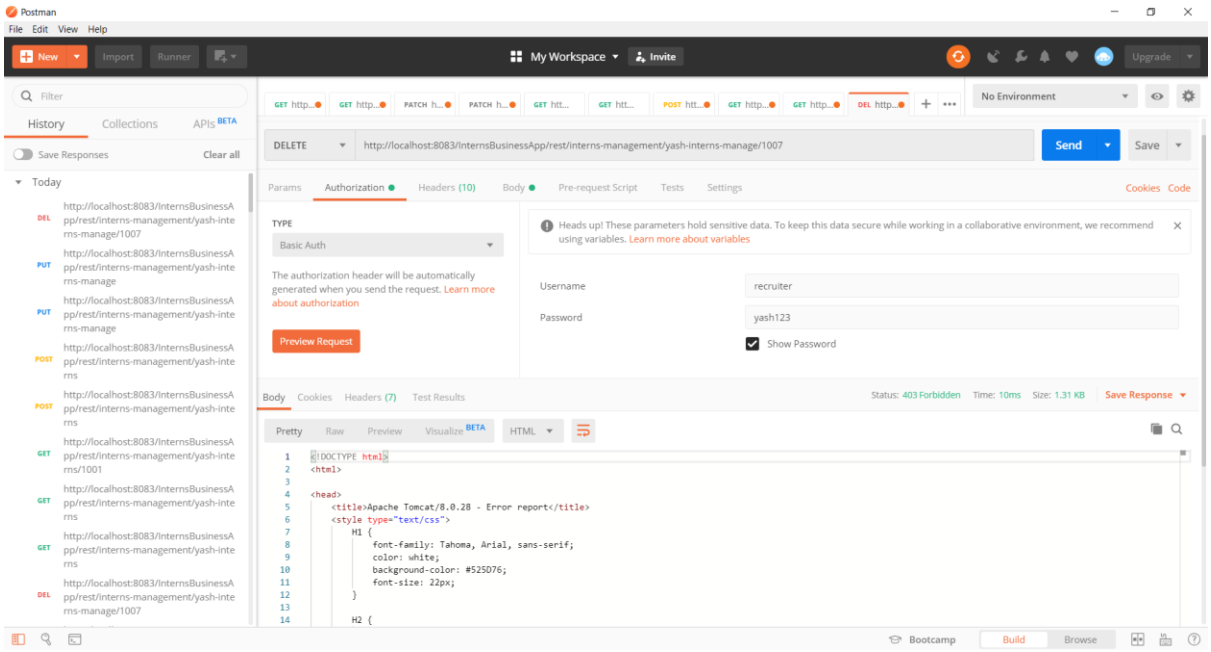


User “recruiter” won’t be able to do above task.

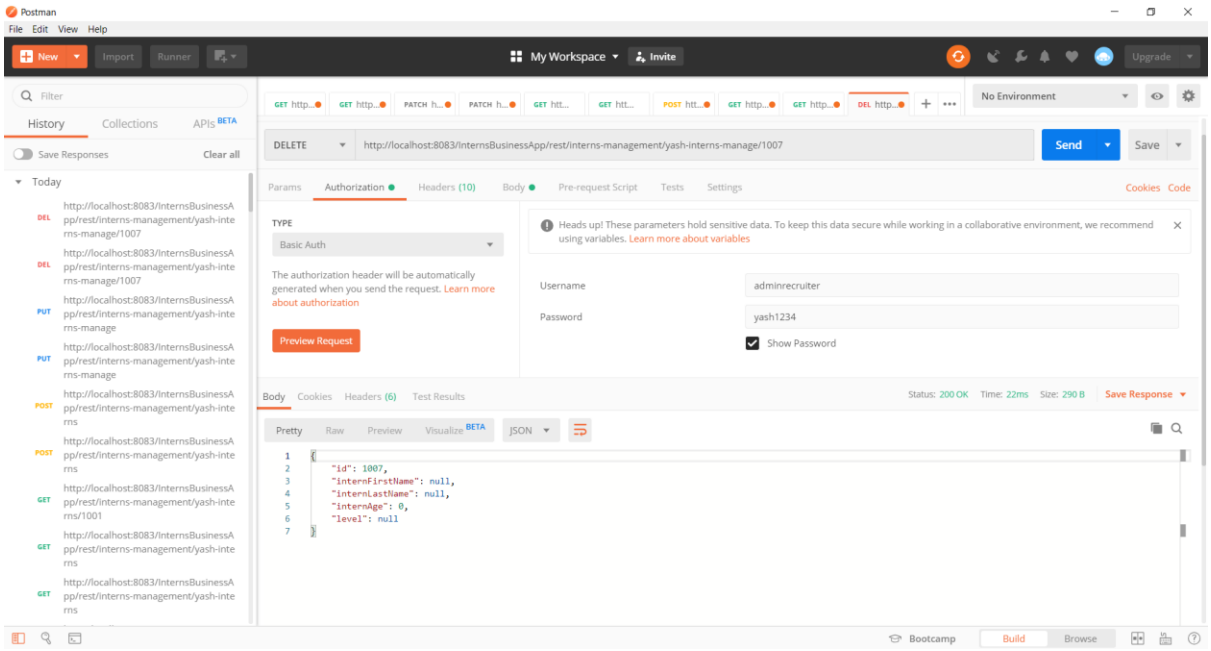


Delete: http://localhost:8083/InternsBusinessApp/rest/interns-management/yash-interns-manage/1007

User with role JuniorRecruiter won’t be able to do above task,



Removal of intern data can be done only by user with role “SeniorRecruiter”



HttpAuthenticationFeature class provides HttpBasic and Digest client authentication capabilities. The feature work in one of 4 modes i.e. BASIC, BASIC NON-PREEMPTIVE, DIGEST and UNIVERSAL. Let’s quickly learn about them.

1. **BASIC** – It’s preemptive authentication way i.e. information is send always with each HTTP request. This mode must be combined with usage of SSL/TLS as the password is send only BASE64 encoded.
2. **BASIC NON-PREEMPTIVE** – It’s non-preemptive authentication way i.e. auth information is added only when server refuses the request with 401 status code and then the request is repeated with authentication information.
3. **DIGEST** – Http digest authentication. Does not require usage of SSL/TLS.
4. **UNIVERSAL** – Combination of basic and digest authentication in non-preemptive mode i.e. in case of 401 response, an appropriate authentication is used based on the authentication requested as defined in WWW-Authenticate HTTP header.

To use HttpAuthenticationFeature, build an instance of it and register with client.

1.1. Basic authentication mode

```
HttpAuthenticationFeature feature = HttpAuthenticationFeature.basic("username", "password");

final Client client = ClientBuilder.newClient();
client.register(feature);
```

1.2. Basic authentication – non-prempitive mode

```
HttpAuthenticationFeature feature = HttpAuthenticationFeature.basicBuilder()
                        .nonPreemptive()
                        .credentials("username", "password")
                        .build();

final Client client = ClientBuilder.newClient();
client.register(feature);
```

1.3. Universal mode

```
//HttpAuthenticationFeature feature =
HttpAuthenticationFeature.universal("username", "password");

//Universal builder having different credentials for different schemes
HttpAuthenticationFeature feature = HttpAuthenticationFeature.universalBuilder()
            .credentialsForBasic("username1", "password1")
            .credentials("username2", "password2").build();

final Client client = ClientBuilder.newClient();
client.register(feature);
```

To test implementation of security in JAX-RS Client we created in earlier section,

In each class in com.yash.client, place below code for Basic Authentication for retrieval of interns data

```
HttpAuthenticationFeature feature = HttpAuthenticationFeature.basic("recruiter",
"yash123");
            Client client = ClientBuilder.newClient( new
ClientConfig().register( LoggingFilter.class ) );

            client.register(feature);
```

or

for invoking register, update and removal service

```
HttpAuthenticationFeature feature =
HttpAuthenticationFeature.basic("adminrecruiter", "yash1234");
            Client client = ClientBuilder.newClient( new
ClientConfig().register( LoggingFilter.class ) );

            client.register(feature);
```