API

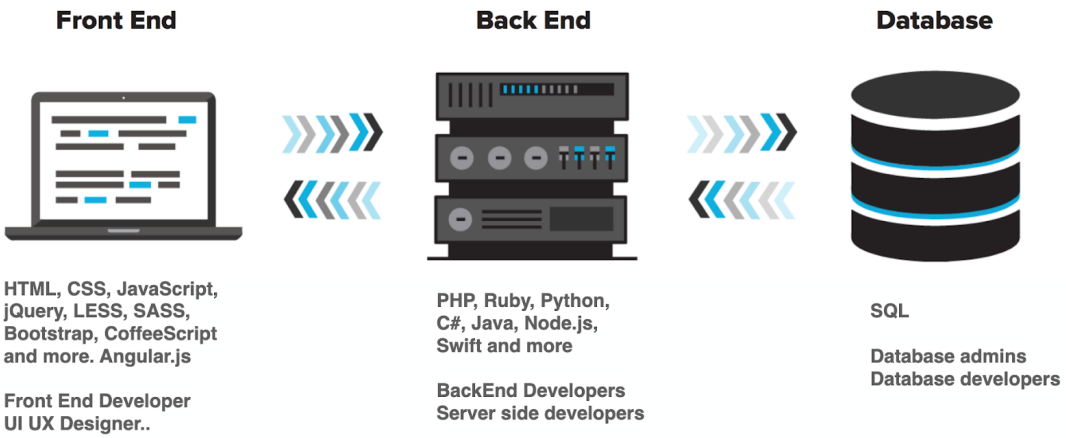
What is API?

- ➔ API stands for Application Programming Interface.
- ➔ It is a set of rules that allow programs to talk to each other. The developer creates the API on the server and allows the client to talk to it.
- ➔ API layer of any application is one of the most crucial software components. It is the channel which connects the client to a server or one microservice to another.

What is a REST API?

- ➔ Let's say you're trying to find videos about Batman on YouTube. You open YouTube, type "Batman" into the search field, hit enter, and you see a list of videos about Batman.
- ➔ A REST API works in a similar way. You search for something and you get a list of results back from the server you're requesting from.
- ➔ **REST** determines how the API looks like. It stands for **"Representational State Transfer"**. It is a set of rules that developers follow when they create their API. One of these rules states that you should be able to get a piece of data (called a **resource**) when you link to a specific URL.
- ➔ Each **URL** is called a **request** when the data sent back to you is called **response**.
- ➔ REST API acts like a business logic layer between the server and front-end layer. It is **independent** of any language as it gets everything in xml/Json format.

Web Application Structure:



Front-End(UI)	⟵RestAPI⟶	Back-End	⟵JDBC⟶	DataBase

- ➔ **Front-End:** User facing web pages to interact with the application.
- ➔ **Back-End:**
  - ◆ **Server Layer:** This is where all business logic code is written in the language of choice.
  - ◆ **DataBase Layer:** This is the database to store the data.

What Is a Web Service?

- Simply put, a web service is a resource that's made available over the internet. Therefore, web services, by definition, require a network.
- Web service is a standardized medium to generate communication between the client and server applications on the World Wide Web.
- Web services provide a common platform that allows multiple applications built on various programming languages to have the ability to communicate with each other.
- Web service talks not only to the UI and database, but also it can talk to other web services to retrieve some information. Like google api, is very commonly used to validate email addresses or log in information in other websites. You can log into eBay with your Gmail where validation is done by Google.
- **Popular Web Services Protocols** are: SOAP, REST, and WSDL.

REST and SOAP

REST	SOAP
➢ Representational State Transfer ➢ An architectural style, or design pattern, for APIs. ➢ Documentation is easy to understand ➢ Requires use of HTTP ➢ More efficient and faster ➢ Requires less bandwidth and resources ➢ Less Secure ➢ REST can use the data JSON, XML, CSV, YAML, plain text, etc ➢ Uses URL to expose business logic ➢ Uses HTTP protocol for communication ➢ Stateless	➢ Simple Object Access Protocol ➢ XML based message protocol ➢ Language, platform, and transport independent, can use SMTP, FTP etc. ➢ Slower compared to REST ➢ Requires more bandwidth and resources ➢ More secure (usually banks use it) ➢ SOAP relies exclusively on XML ➢ Uses services interfaces to expose business logic ➢ SOAP is a protocol ➢ Stateful

What is the advantage of using REST?

- REST allows a greater variety of data formats, whereas SOAP only allows XML.
- Coupled with JSON (which typically works better with data and offers faster parsing), REST is generally considered easier to work with.
- Thanks to JSON, REST offers better support for browser clients.
- REST provides superior performance, particularly through caching for information that's not altered and not dynamic
- It is the protocol used most often for major services such as Yahoo, Ebay, Amazon, and even Google.
- REST is generally faster and uses less bandwidth. It's also easier to integrate with existing websites with no need to refactor site infrastructure. This enables developers to work faster rather than spend time rewriting a site from scratch. Instead, they can simply add additional functionality.

Stateless vs Stateful Protocols

Stateless	Stateful
Stateless Protocol does not require the server to retain the server information or session details.	Stateful Protocol requires the server to save the status and session information.
There is no tight dependency between server and client.	There is tight dependency between server and client.
It works better at the time of crash because there is no state that must be restored, a failed server can simply restart after a crash.	It does not work better at the time of crash because stateful servers have to keep the information of the status and session details of the internal states.
Stateless Protocols handle the transaction very fastly.	Stateful Protocols handle the transaction very slowly.
Stateless Protocols are easy to implement on the Internet.	Stateful protocols are logically heavy to implement on the Internet.

HTTP Methods for RESTful APIs

HTTP METHOD	CRUD	ENTIRE COLLECTION (E.G. /USERS)	SPECIFIC ITEM (E.G. /USERS/123)
POST	Create	201 (Created), 'Location' header with link to /users/{id} containing new ID.	Avoid using POST on single resource
GET	Read	200 (OK), list of users. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single user. 404 (Not Found), if ID not found or invalid.
PUT	Update/ Replace	405 (Method not allowed), unless you want to update every resource in the entire collection of resources.	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID is not found or invalid.
PATCH	Partial Update/ Modify	405 (Method not allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID is not found or invalid.
DELETE	Delete	405 (Method not allowed), unless you want to delete the whole collection — use with caution.	200 (OK). 404 (Not Found), if ID not found or invalid.

What is EndPoint?

<protocol>://<service-name>/<ResourceType>/ResourceID → URI (Uniform Resource Identifier)
Base URI / resource ? Parameters

(http://www.google.com/search?source=book...) -- ? -- query parameters

Query Parameter vs Path Parameter

- ➔ If you want to identify a resource, you should use Path Variable. But if you want to sort or filter items, then you should use a query parameter.
- ➔ Path parameter stands for fetching specific resources, like: some city info, user, etc..
- ➔ Query parameters stand for filtering purposes. For example, out of all users, we need a user with the last name Bond, or a user that has a masters degree so we use query param here.

HTML Status Codes?

- ➔ **1xx** → Informational
- ➔ **2xx** → Success (request was accepted successfully)
  - ◆ 200→ Ok
  - ◆ 201→ Created
  - ◆ 202→ Accepted
  - ◆ 204→ No Content
- ➔ **3xx** → Redirection
- ➔ **4xx** → Client Error
  - ◆ 400-Bad Request
  - ◆ 401-Unauthorized
  - ◆ 403-Forbidden
  - ◆ 404-Not Found
  - ◆ 405-Method not Allowed
- ➔ **5xx** → Server Error
  - ◆ 500-Internal server Error
  - ◆ 502-Bad Gateway
  - ◆ 501-Not implemented
  - ◆ 503-Service Unavailable

What first thing you check when you get a response?

- ➔ Status quote (200 always mean Ok)
- ➔ We always check the 404 means not found
- ➔ rest-assured.io==> for automation to find the EC2 machine in search type remote Desktop

What is URI, purpose and format?

- ➔ URI stands for Uniform Resource Identifier.
- ➔ The purpose of URI is to locate a resource on the server hosting the web service.
- ➔ A URI is of the following format:
  <protocol>://<service-name>/<ResourceType>/<ResourceID>

What WebServices do you use in your project?

- I use Restful which is Representational State of Transfer and it communicates with XML and JSON, but my current project uses JSON

What is XML?

- In computing, Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

What is JSON?

- It is JavaScript Object Notation (is a minimal, readable format for structuring data.)
- It is used primarily to transmit data between a server and web application, as an alternative to XML.
- A lightweight version of XML
- In Key: Value format
  Key is always in double quotes and value if string its double quotes and if numbers no quotes
- It is purely based on http protocol, - so it hits the link on the browser and see the results
- JSON is based on JavaScript. The main difference from a JavaScript file is that JSON does not really stand for any scripting. It is not executable.
- It contains states but not methods. It can have different property types like: String, Boolean, Integer. It can have Arrays and Objects.
- JSON it's a data representational language, not a programming language. JavaScript natively supports JSON, because obviously JSON is based on JavaScript. JSON is used for transferring data between server and client, client and server, server and server, etc...
- JSON is very light weight, language independent, easy to read and parse, plain text so it's human readable
- Most of the languages have libraries that can parse from JSON to object (De-serialization) or from object to JSON (Serialization).

What is JsonPath?

- Another way to validate response body
- It is an alternative to using XPath for easily getting the values from an object document.
- used to navigate in a json document. Rest assured we can use the jsonpath as an operating object or as part of the body() method to extract values and verify.
- JsonPath jasonPath=response.jsonpath();

Do You Know Swagger? What's Swagger?

- Swagger is an open-source software framework backed by a large ecosystem of tools that helps developers design, build, document, and consume RESTful Web services.
- Swagger allows you to describe the structure of your APIs so that machines can read them.
- The ability of APIs to describe their own structure is the root of all awesomeness in Swagger → Similar to xml schema but for Json

json vs gson

- ➔ JSON is a format which has key and values.
- ➔ Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object.
- ➔ GSON is a process of converting:
    - ◆ from java to json(serialization)
    - ◆ from json to java(deserialization)

What are HTTP Request and HTTP Response?

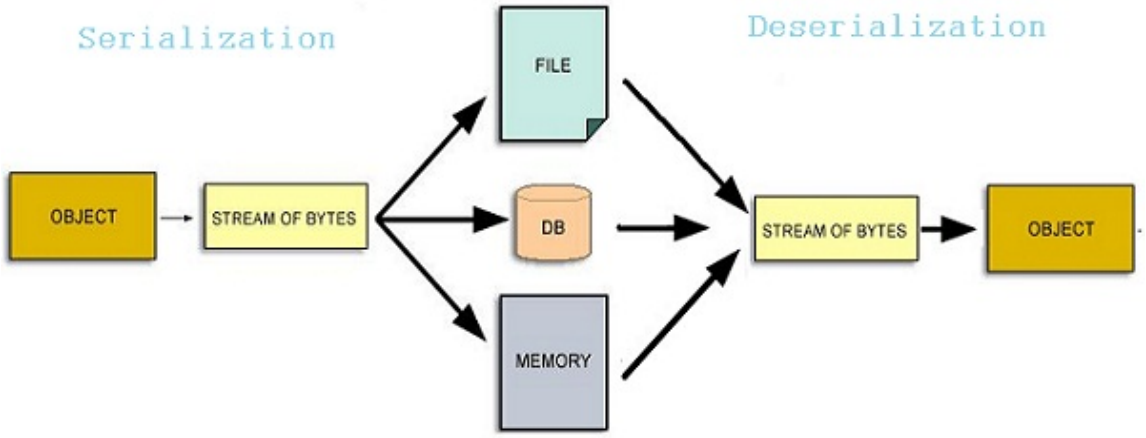
HTTP request is what the client sends to the server. It has detailed information about what we want. Requests have urls, methods, headers, parameters, body. Request is made up of **four** components:

- ➔ **Request Method:** Get, Post, Put, Delete (these are the common ones)
- ➔ **Request URI:** the URL of the resource
- ➔ **Request Header:** Accept-Language, Accept-Encoding, User-Agent, Host
- ➔ **Request Body:** This is the data to be sent to the resource.

HTTP response is what we get back from the server as a result/answer/reply. Response has status code, headers, and body. Itis made up of **three** components:

- ➔ Response Status Code: 200, 301, 404, 500 (these are the most common ones)
- ➔ Response Header Fields: Date, Server, Last-Modified, Content-Type
- ➔ Response Body: This is the data that comes back to the client from the server

Serialization and Deserialization



Gson is a Java serialization/deserialization library to convert Java Objects into JSON and back. Gson was created by Google for internal use and later open sourced.

Every language has their own library(libraries) to perform Serialization and Deserialization, In java we have Gson, Jackson, etc. In my project, we used Gson. Serialization and Deserialization not automatically means pojo->json json->pojo.

It can be anything, the main idea is converting pojo into streams of bytes and opposite.

Serialization:

- ➔ Serialization in the context of Gson means mapping a Java object to its JSON representation.
- ➔ When we MAP a Java object to API JSON format (CONVERT JAVA OBJECT TO JSON);
  - ◆ Java object (POJO(Plain Old Java Object), BEANS) → MAP it to API JSON/XML
  - ◆ When we have an object from a class and MAP it to a JSON format in our RESTful API

```
{make: “Toyota”,
Model: “Camry” }
Car car = new Car();
car.setMake(“Toyota”);
car.setModel(“Camry”);
given().body(car).when().post(uri)
```

De-Serialization:

- ➔ It is converting a json file into Java Object.
- ➔ Deserialization: API JSON/XML → MAP it to Java Object (JSON TO JAVA OBJECT)

```
Car car2 = new Car();
car2=when().get(uri).body.as(car.class);
car.setMake(“Toyota”);
car.setModel(“Camry”);
```

POJO

- ➔ POJO stands for Plain Old Java Object. It is an ordinary Java object, not bound by any special restriction other than those forced by the Java Language Specification and not requiring any class path. POJOs are used for increasing the readability and re-usability of a program.
- ➔ POJOs basically defines an entity. Like in your program, if you want an Employee class then you can create a POJO.
- ➔ All properties private (use getters/setters; A public no-argument constructor; Implements Serializable.
- ➔ POJO is an object which encapsulates Business Logic. Following image shows a working example of POJO class. Controllers interact with your business logic which in turn interact with POJO to access the database. In this example a database entity is represented by POJO. This POJO has the same members as the database entity.

```
public class Room {
    private int id;
    private String name;
    private String description;
    private int capacity;
    private boolean withTV;
    private boolean withWhiteBoard;

    public Room(){

    }

    public Room(String name, String description, int capacity, boolean withTV, boolean withWhiteBoard) {
        this.name = name;
        this.description = description;
        this.capacity = capacity;
        this.withTV = withTV;
        this.withWhiteBoard = withWhiteBoard;
    }


public int getId() { return id; }
public void setId(int id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public String getDescription() { return description; }
public void setDescription(String description) { this.description = description; }
public int getCapacity() { return capacity; }
public void setCapacity(int capacity) { this.capacity = capacity; }
public boolean isWithTV() { return withTV; }
public void setWithTV(boolean withTV) { this.withTV = withTV; }
public boolean isWithWhiteBoard() { return withWhiteBoard; }
public void setWithWhiteBoard(boolean withWhiteBoard) { this.withWhiteBoard = withWhiteBoard; }

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Room)) return false;
    Room room = (Room) o;
    return id == room.id &&
            capacity == room.capacity &&
            withTV == room.withTV &&
            withWhiteBoard == room.withWhiteBoard &&
            name.equals(room.name) &&
            description.equals(room.description);
}
```

```
@Override
public int hashCode() { return Objects.hash(id, name, description, capacity, withTV, withWhiteBoard); }

@Override
public String toString() {
    return "Room{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", description='" + description + '\'' +
            ", capacity=" + capacity +
            ", withTV=" + withTV +
            ", withWhiteBoard=" + withWhiteBoard +
            '}';
}
```

What is Hamcrest Matcher for?

- ➔ Hamcrest is a framework for software tests. Hamcrest allows checking for conditions in your code via existing matchers classes. It also allows you to define your custom matcher implementations.
- ➔ I use Hamcrest matchers with JUnit. You use the assertThat() statement followed by one or several matchers.
- ➔ Hamcrest is typically viewed as a third generation matcher framework. The first generation used assert(logical statement) but such tests were not easily readable. The second generation introduced special methods for assertions, e.g., assertEquals(). This approach leads to lots of assert methods. Hamcrest uses assertThat method with a matcher expression to determine if the test was successful.

```
import org.junit.jupiter.api.Test;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;

public class BiscuitTest {
  @Test
  public void testEquals() {
    Biscuit theBiscuit = new Biscuit("Ginger");
    Biscuit myBiscuit = new Biscuit("Ginger");
    assertThat(theBiscuit, equalTo(myBiscuit));
   assertThat("chocolate chips", theBiscuit.getChocolateChipCount(), equalTo(10));
   assertThat("hazelnuts", theBiscuit.getHazelnutCount(), equalTo(3));
  }
}
```

```
// verify if first argument is equal to the second
assertThat(str1, is("Kunkka"));
assertThat(str1, is(str2));

// verify if first argument is NOT equal to the second
assertThat(str1, is(not("Tidehunter")));

// compare ignoring case
assertThat(str1, equalToIgnoringCase("kunkka"));

// compare ignoring space before and after
assertThat(str1, equalToIgnoringWhiteSpace(" Kunkka "));

// compare numbers
assertThat(10, greaterThan(9));
assertThat(10, lessThan(11));
assertThat(10, lessThanOrEqualTo(11));

// verify not null
assertThat(str1, notNullValue());

List<String> list = Arrays.asList("one", "too", "tree");
   assertThat(list, hasSize(3));
   assertThat(list, containsInAnyOrder("too", "tree", "one"));
   assertThat(list, hasItems("one", "too"));

List<Integer> numbers = Arrays.asList(11, 12, 13);
assertThat(numbers, everyItem(greaterThan(9)));
```

Authorization vs Authentication

- ➔ authentication --> who are you
- ➔ authorization --> what rights do you have
- ➔ **Authentication**:
  It is the act of validating that users are who they claim to be. Passwords are the most common authentication factor—if a user enters the correct password, the system assumes the identity is valid and grants access.
  - ◆ **Single- Factor Authentication:** This is the simplest form of authentication method which requires a **password** to grant user access to a particular system such as a website or a network. In selenium, we used to enter **username** and **password** like this (for basic authentication)
  - ◆ **Two- Factor Authentication:** This authentication requires a two- step verification process which not only requires a **username and password,** but also a piece of **information** only the user knows.
  - ◆ **Multi- Factor Authentication:** This is the most advanced method of authentication which requires two or more levels of security from independent categories of authentication to grant user access to the system. This form of authentication utilizes factors that are independent of each other in order to eliminate any data exposure. It is common for financial organizations, banks, and law enforcement agencies to use multiple- factor authentication.

➔ **Authorization** :
Authorization occurs after your identity is successfully authenticated by the system, which therefore gives you full access to resources such as information, files, databases, funds, etc. However authorization verifies your rights to grant you access to resources only after determining your ability to access the system and up to what extent. In other words, authorization is the process to determine whether the authenticated user has access to the particular resources.

- No Authorization
- **Basic Authentication:**
  HTTP Basic Authentication is rarely recommended due to its inherent security vulnerabilities. This is the most straightforward method and the easiest. With this method, the sender places a **username**:**password** into the request header. The username and password are encoded with Base64, which is an encoding technique that converts the username and password into a set of 64 characters to ensure safe transmission.
  This method does not require cookies, session IDs, login pages, and other such specialty solutions, and because it uses the HTTP header itself, there's no need to handshakes or other complex response systems.
  Authorization: Basic bG9sOnNlY3VyZQ==
- **Bearer Token:** Bearer authentication (also called token authentication) is an HTTP authentication scheme that involves security tokens called bearer tokens.
  The name "Bearer authentication" can be understood as "give access to the bearer of this token." The bearer token allows access to a certain resource or URL and most likely is a cryptic string, usually generated by the server in response to a login request.

  The client must send this token in the Authorization header when making requests to protected resources: Authorization: Bearer <token>

  key = Authorization
  value = Bearer
  eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiI1Mzc2IiwiYXVkIjoic3R1ZGVudC10ZWFtLWxlYWRlciJ9.DoFI744aMLxUaf0GcjVOEDkJ3Wh7RlKDx-TYp8_sJpU

- **API Keys:**
  In REST API Security - API keys are widely used in the industry and have become some sort of standard, however, this method should not be considered a good security measure.
  In this method, a unique generated value is assigned to each first time user, signifying that the user is known. When the user attempts to re-enter the system, their unique key (sometimes generated from their hardware combination and IP data, and other times randomly generated by the server which knows them) is used to prove that they're the same user as before.
  Many API keys are sent in the query string as part of the URL, which makes it easier to discover for someone who should not have access to it.
  Authorization: Apikey 1234567890abcdef.
- **OAuth (2.0):**
  The previous versions of this spec, OAuth 1.0 and 1.0a, were much more

complicated than OAuth 2.0. The biggest change in the latest version is that it's no longer required to sign each call with a keyed hash. The most common implementations of OAuth use one or both of these tokens instead:

access token: sent like an API key, it allows the application to access a user's data; optionally, access tokens can expire.

refresh token: optionally part of an OAuth flow, refresh tokens retrieve a new access token if they have expired. OAuth2 combines Authentication and Authorization to allow more sophisticated scope and validity control.

OAuth 2.0 is the best choice for identifying personal user accounts and granting proper permissions. In this method, the user logs into a system. That system will then request authentication, usually in the form of a token. The user will then forward this request to an authentication server, which will either reject or allow this authentication. From here, the token is provided to the user, and then to the requester. Such a token can then be checked at any time independently of the user by the requester for validation and can be used over time with strictly limited scope and age of validity.
OAuth 2.0. allows authentication via 3rd party services, without sharing your password.

RestAssured Log Logging Logs:

- **Request Logging**

```
given().log().all() // Log all request specification details including parameters, headers and body
given().log().params() // Log only the parameters of the request
given().log().body() // Log only the request body
given().log().headers() // Log only the request headers
given().log().cookies() // Log only the request cookies
given().log().method() // Log only the request method
given().log().path() // Log only the request path
```

- **Response Logging**

```
get("/x").then().log().body()
get("/x").then().log().ifError()
get("/x").then().log().all()
get("/x").then().log().statusLine() // Only log the status line
get("/x").then().log().headers() // Only log the response headers
get("/x").then().log().cookies()  // Only log the response cookies
get("/x").then().log().ifStatusCodeIsEqualTo(302)
   // Only log if the status code is equal to 302
get("/x").then().log().ifStatusCodeMatches(matcher)
   // Only log if the status code matches the supplied Hamcrest matcher
```

API test strategy:

API testing involves APIs directly and checks whether the API meets expectations in terms of **functionality**, **reliability**, **performance**, and **security** of an application. My first concern is **functional testing** which ensures that the API functions correctly.

The main objectives in functional testing of the API are:

- ➔ to ensure that the implementation is working correctly as expected - no bugs!
- ➔ to ensure that the implementation is working as specified according to API documentation.
- ➔ to prevent regressions between code merges and releases.

I have 4 different process to implement: I HAVE FOUR DIFFERENT PROCESS TO IMPLEMENT

1. Checking API contract
2. Creating test cases
3. Executing test cases
4. Implementing different test flows

1. Checking API Contract:

An API is essentially a contract between the client and the server or between two applications. Before any implementation test can begin, it is important to make sure that the contract is correct.

- Endpoints are correct.
- Resource correctly reflects the object model (proper JSON/XML structure used in response),
- There is no missing functionality or duplicate functionality,
- Relationships between resources are reflected in the API correctly.

Now, that we have verified the API contract, we are ready to think of what and how to test.

We use Swagger documentation in our project and I test the endpoint with the try-out button to make sure they are running. Click on me (:

2. Creating test cases:

I mostly create the following test case groups:

- Basic positive test (happy paths)
- Extended positive testing with optional parameters (optional parameters and extra functionality)
- Negative testing with valid input (trying to add an existing username)
- Negative testing with invalid input (trying to add a username which is null)
- Destructive testing (sending null, empty string, integer or other types, odd date format, deleting necessary parameters)
- Security, authorization, and permission tests (sending valid or invalid access tokens to permitted or unpermitted endpoints)

3. Executing test cases:

For each API request I need to verify following items:

- **Data accuracy:** Check the request and response body whether those are as written on API documentation in terms of data type and data structure.

- **HTTP status code:** For example, creating a resource should return 201 CREATED and unpermitted requests should return 403 FORBIDDEN, etc.
- **Response headers:** HTTP server headers have implications on both security and performance.
- **Response body:** Check valid JSON body and correct field names, types, and values - including in error responses.
- **Authorization checks:** Check authentication and authorization
- **Error messages:** Check the error code coverage in case API returns any error
- **Response time:** Implementation of response timeout

4. Test flows:

We need to implement the next test flow if previous flow is success:

- **Single-step workflow:** Executing a single API request and checking the response accordingly. Such basic tests are the minimal building blocks we should start with, and there's no reason to continue testing if these tests fail.
- **Multi-step workflow with several requests:** For example, we execute a POST request that creates a resource with id and we then use this id to check if this resource is present in the list of elements received by a GET request. Then we use a PATCH endpoint to update new data, and we again invoke a GET request to validate the new data. Finally, we DELETE that resource and use GET again to verify it no longer exists.
- **Combined API and UI test:** This is mostly relevant to manual testing, where we want to ensure data integrity between the UI and API. We execute requests via the API and verify the actions through the UI or vice versa. The purpose of these integrity test flows is to ensure that although the resources are affected by different mechanisms the system still maintains expected integrity and consistent flow.

Types of Bugs that API testing detects

- ➔ Fails to handle error conditions gracefully
- ➔ Unused flags
- ➔ Missing or duplicate functionality
- ➔ Reliability Issues. Difficulty in connecting and getting a response from API
- ➔ Security Issues
- ➔ Multi-threading issues
- ➔ Performance Issues. API response time is very high
- ➔ Improper errors/warning to a caller
- ➔ Incorrect handling of valid argument values
- ➔ Response Data is not structured correctly (JSON or XML)

Challenges in API Testing

Some of the challenges we face while doing API testing are as follows

- ➔ Selecting proper parameters and its combinations
- ➔ Categorizing the parameters properly
- ➔ Proper call sequencing is required as this may lead to inadequate coverage in testing
- ➔ Verifying and validating the output
- ➔ Due to absence of GUI it is quite difficult to provide input values

How And Where Are You Sending Requests?

➔ Since I am using Rest, it has endpoints. My developers create public URLs and requests are sent to that URL.

Do you use any non-web services API?

➔ I use Selenium API for browser, JDBC for database, and RestAssured for API

Do you have an API documentation website for your API?

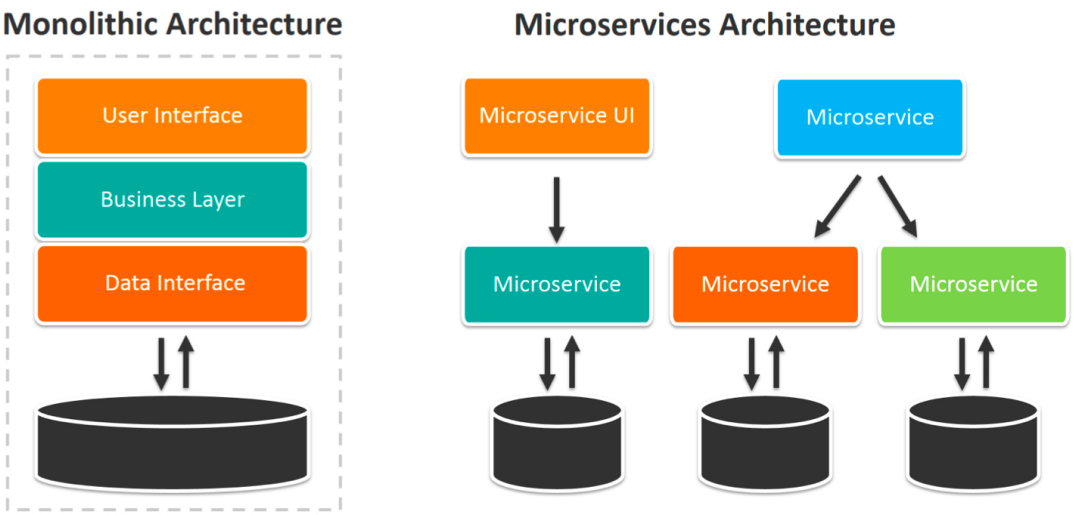
➔ Yes, we use swagger for our api documentation, and this is where the description and guidelines of API endpoints are.

Microservices:

➔ Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are
- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities
- Owned by a small team

➔ The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.

➔ Microservice is an architectural style for building applications. In microservices, application is broken down to smaller pieces/functionalities/services which act as separate/stand alone service.



➔ Using microservices enables changing one service without affecting another one. Microservices act as individual applications/services and typically they have their own database.

As a tester what does it mean to test micro services?

- ➔ Depending on the size of the project we can work on a team that only works on one microservice. Or we can be in a team that handles multiple microservices.
- ➔ Microservices are just web services. So we treat them like any other web service while testing. To test microservices we need endpoints, api documentation, test cases/ requirements and use tools like postman, rest assured, karate, etc.
- ➔ In microservices, we do a lot of mocking. Mocking is imitating a service. We can do mocking with postman, karate, etc.

Can All API endpoints use all of the Http protocols?

- ➔ It depends, My API developer decides if that URL works with GET,POST,PUT, or DELETE requests.

How do you manually test your API?

- ➔ I use Postman → it is a REST API client tool that test the REST API URL

API/Webservices with RestAssured Library?

```
import static io.restassured.RestAssured.* ;
URI uri = new URI(“ ... / methods(get,post)”)
```

- GET;

```
Response response = given().accept(ContentType.JSON).when().get(URI);
response.then().assertThat().statusCode(200).
   and().assertThat().ContentType(ContentType.JSON);
```

- POST;

```
Response response = given().ContentType(ContentType.JSON).with().accept(ContentType.JSON)
.and().body(JSONbody).when().post(URI);
response.then().assertThat().statusCode(200);
```

-

```
import static org.hamcrest.Matchers.* ;
then().assertThat().body(“Id”,Matchers.equalTo(123));
```

-

```
JsonPath json = JsonPath(JSONbody);
json.getString(“key”);
json.getInt(“key”);
json.getList(“key1.key2”);
```

How are you using Enum in your project?

- ➔ I am using content Type to make sure that my response type is JSON format

How did you do UI, API and Db testing in your framework?

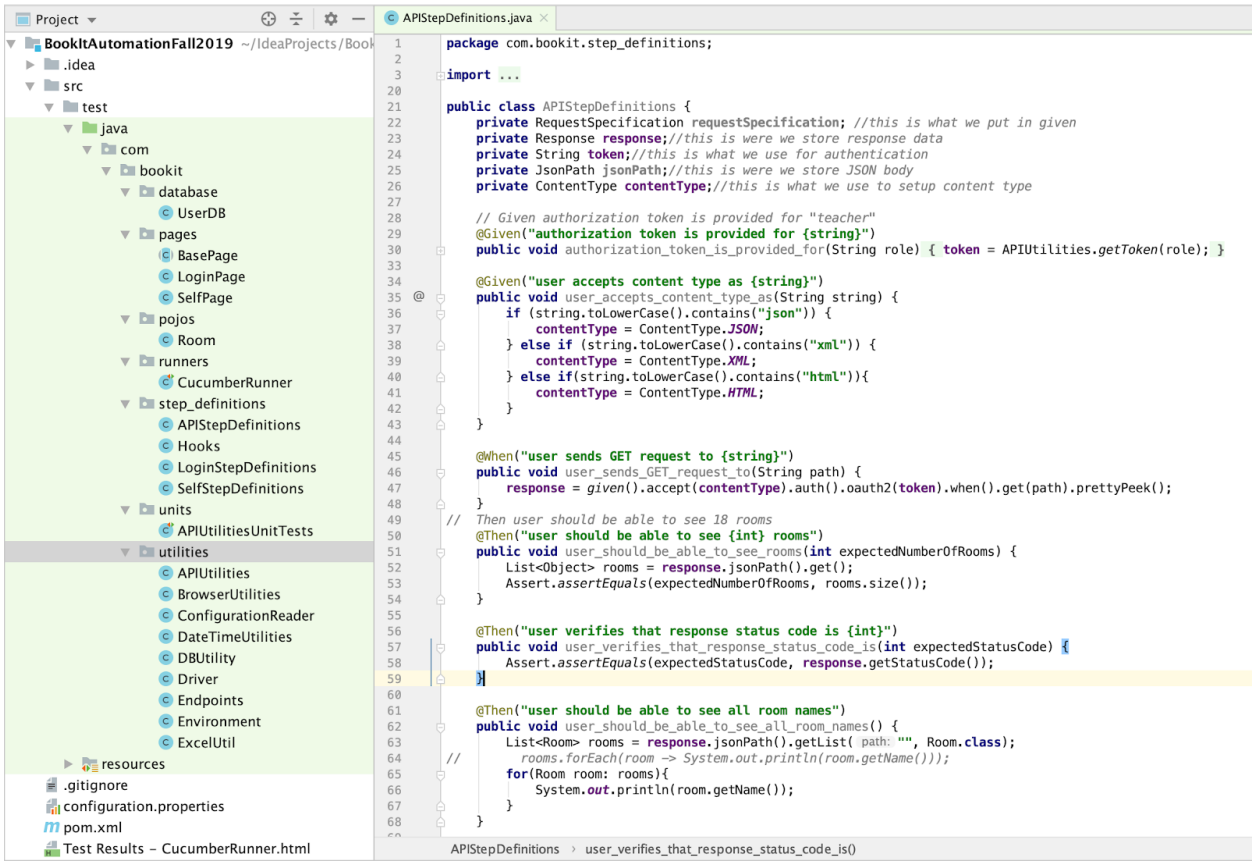
- ➔ In my current project I have tests with ui, api and db.
- ➔ In some test cases even though I did not have API in test steps, I used api for test data creation.
- ➔ I used api, and db when cleaning up the database after tests.
- ➔ I used Selenium for UI, rest assured for api, jdbc for database.

How did you handle environments in your projects?

1. I had an Environment utility class which helped me handle the information related to 3 different environments. This class loads the information specific to each information from the properties file for that environment. I can pass the environment type form the main configuration.properties file or I can pass it from the command line.
2. I created steps to auto generate test data using api. This way my test cases don't have dependency on environment specific test data.

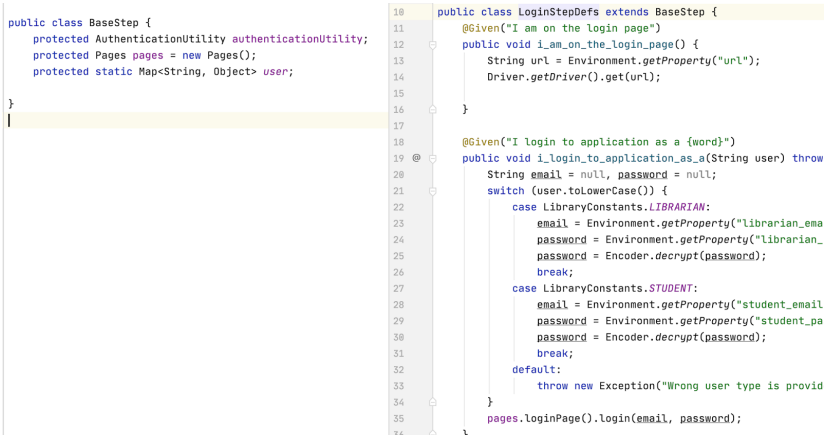

```
package com.cybertek.library.utilities.common;


import ...

public class Environment {
    private static Properties properties;

    static {
        try {
            // LOAD GENERAL PROPERTIES
            String path = "configuration.properties";
            FileInputStream input = new FileInputStream(path)

            properties = new Properties();
            properties.load(input);

            // LOAD ENVIRONMENT SPECIFIC PROPERTIES
            if (System.getProperty("env") != null) {
                path = "src/test/resources/env/" + System.get
            } else {
                path = "src/test/resources/env/" + properties
            }
```

BookIt Project Structure



How did you use OOPS Concepts in your project?

Inheritance: Page object classes extend the BasePage which has the common elements



Abstraction:

```
public abstract class BaseStep {
    protected AuthenticationUtility authenticationUtility;
    protected Pages pages = new Pages();
    protected static Map<String, Object> user;
```

```
public abstract class BasePage {
    public BasePage() { PageFactory.initElements(Driver.getDriver(), page: this); }

    @FindBy(xpath = "//span[@class='title'][.='Users']")
    public WebElement users;

    @FindBy(xpath = "//span[@class='title'][.='Dashboard']")
    public WebElement dashboard;

    @FindBy(xpath = "//span[@class='title'][.='Books']")
    public WebElement books;

    @FindBy(tagName = "h3")
    public WebElement pageHeader;

    @FindBy(css = "#navbarDropdown>span")
    public WebElement accountHolderName;

    @FindBy(linkText = "Log Out")
    public WebElement logOutLink;

    public void logOut(){
        accountHolderName.click();
```

Encapsulation: • Pojos/beans, Pages class, • Page object classes, • Driver class

```
public class Pages {
    private DashBoardPage dashBoardPage;
    private LoginPage loginPage;
    private UsersPage usersPage;
    private BooksPage booksPage;

    public Pages() {
        this.dashBoardPage = new DashBoardPage();
        this.loginPage = new LoginPage();
        this.usersPage = new UsersPage();
        this.booksPage = new BooksPage();
    }

    public DashBoardPage dashBoardPage() { return dashBoardPa

    public LoginPage loginPage() { return loginPage; }

    public UsersPage usersPage() { return usersPage; }

    public BooksPage booksPage() { return booksPage; }
```

```
public class Book {
    private String name;
    private String author;
    private String year;
    private String category;
    private String isbn;
    private String description;

    public Book(String name, String author, String ye
        this.name = name;
        this.author = author;
        this.year = year;
    }

    public String getName() { return name; }

    public void setName(String name) { this.name = na

    public String getAuthor() { return author; }

    public void setAuthor(String author) { this.autho

    public String getYear() { return year; }

    public void setYear(String year) { this.year = ye
```

Polymorphism • I use an interface for AuthenticationUtility. Of each type of user, I have a different implementation of that interface AuthenticationUtility.

```java
public interface AuthenticationUtility {

    Response getLoginResponse();

    String getToken();

    String getRedirectUrl();
}
```

```java
public class LibrarianAuthenticationUtility implements AuthenticationUtility {
    private static Response response;
    private String token;
    private String redirectUrl;

    @Override
    public Response getLoginResponse() {
        if (response == null) {
            String username = Environment.getProperty("librarian_email");
            String password = Environment.getProperty("librarian_password");
            password = Encoder.decrypt(password);
            response = given().
                    formParam( parameterName: "email", username).
                    formParam( parameterName: "password", password).
                    log().all().
                when().
```

How to pass command line arguments?

Password in this comes from the terminal. It means every time we run the test, we have to pass the password. Otherwise it will fail.

```java
String password = System.getProperty("db_password");
// mvn test -Ddb_password=abc123
```

How did you handle sensitive data in your framework?

1. I passed sensitive data like database username and password from the terminal.
2. We used password encoders. We have the encoded version of the password saved in the properties file. In the code we decode the password before using it.

Karate API Testing

Karate is the only open-source tool to combine API test-automation, mocks, performance-testing and even UI automation into a single, unified framework. The BDD syntax popularized by Cucumber is language-neutral, and easy for even non-programmers. Powerful JSON & XML assertions are built-in, and you can run tests in parallel for speed.

```
Scenario: create and retrieve a cat

Given url 'http://myhost.com/v1/cats'
And request { name: 'Billie' }
When method post
Then status 201
And match response == { id: '#notnull', name: 'Billie' }

Given path response.id
When method get
Then status 200
```

JSON is 'native' to the syntax

Intuitive DSL for HTTP

Payload assertion in one line

Second HTTP call using response data