QA Approach for Rest API Testing

Scenario: You need to establish a QA process in a cross-functional team. The team builds a front-end application using REST APIs.

Question: Where would you start? What would be your first steps?

Always the first step of QA process would be gathering, understanding and analyzing of requirements. In case of front end development through REST API, I would follow the below strategy

Understanding on a high level: First we need to understand what kind of front end application are we developing? What is the business functionality? Which systems are included and integrated? What is the actual scope of QA testing?

1. I would start **involving in feature planning and wireframe/prototype sessions** along with BA and Development team, that would help me in understanding the requirements and application we are building on a high level.
2. I will **provide my feedback on prototype designs to development and BA team**, this kind of early feedback will help them improvise their design, reduce gaps and help in defect prevention.

Gathering requirements: Once the requirements are finalized. BA team will be ready to share the documented business requirements and development team will be sharing the API interface specification documents/detailed level design documents/ UML diagrams/process flow diagrams/MS Visio's

1. **Best practice for REST API development is to maintain a knowledge base for specifications**. I prefer developers to use **Confluence page to document interface specification** details of API. So API specifications are easily accessible and maintained across cross functional team
2. I suggest Development team to **maintain version control on API specification page**
3. As a QA resource, I accept the below mentioned details in the REST API interface specifications for each API method
    a. API method description
    b. URI
    c. Methods (Post/Put/Get/delete)
    d. Request and response parameters
    e. Sample request
    f. Error codes (System generated; user defined)

Brain storming (Requirement analysis) sessions with QA team: Once all the required specifications/ requirements are gathered. Then I will schedule recurring meeting with the QA team to analyze the requirements.

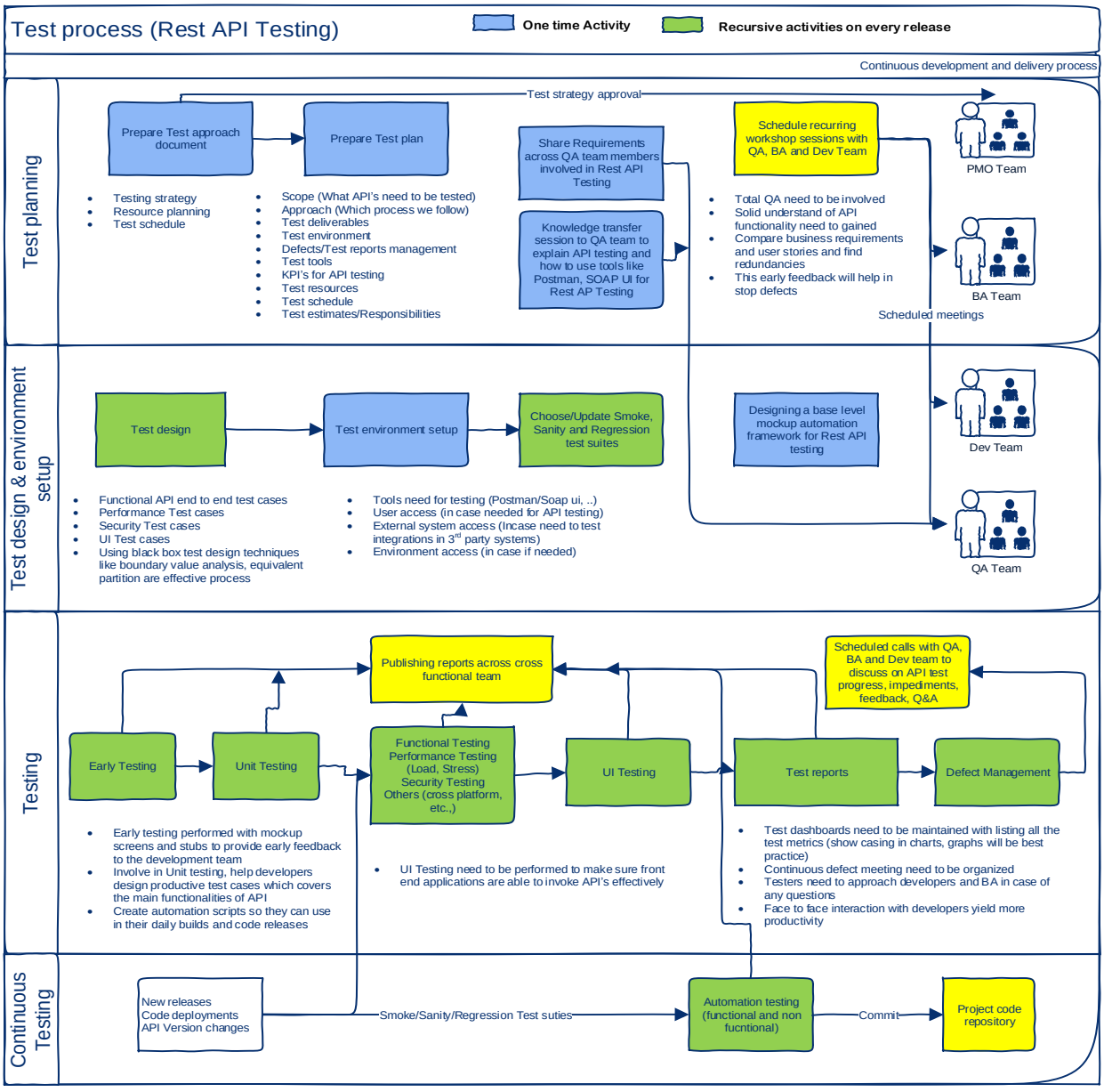
1. Meeting agenda would be, QA team understanding requirements, **get clarity of application being developed, find redundancies** and verifiability
2. I prefer to explain the API flows with the help of white board drawings
3. I will **document meeting notes** from the inputs of team members. I will redirect their questions and feedback to BA & Dev team. I will try to extract and **provide clarifications to the team**

4. Requirement analysis will be well documented and maintained in test management applications (Test rail, ALM, etc.,) or on Confluence pages

Question: Which process would you establish around testing new functionality?

Every Test process should follow STLC, but the implementation of the STLC will vary depending on what kind of software methodology (Agile, V model) we are using to develop the application. Irrespective of the type of application and software methodology, **Test process should always be focused on 3 major principles: Test early, Test often and Test continuously**

Rather than explaining it in long sentences, I have drawn the below REST API test process flow diagram so that you can quickly absorb my approach. (Please zoom in if you are not able to view the image)



Question: How would you want the features to be tested?

The testing of Rest API's will be continuous and repetitive throughout the application development. API's need to be tested on below test levels and types

API Test levels:

- **Early Testing**: This testing is done by QA **team using mock ups screens and stubs**. The main motive of this testing is to provide feedback to developers before coding and also QA team will gain hands-on experience with API's
- **Unit testing**: Done by developers. But **QA team will provide support to developers** in test design to make sure major functionalities of API are covered in unit test cases
- **Integration Testing**: QA team tests to confirm Rest API is **able to integrate, communicate and pass information between components** client server applications. This is performed without any UI
- **System Testing**: End to end testing of API functionality, **all possible test cases need to be tested**, including negative and positive test cases. Different status code validations (200, 400, etc.,) should also need to be invoked

Types of Tests:

- **Functional**: End to end testing of API to make user the API has been developed and working as per the business requirements of front end application
- **Non functional**:
  - **Performance testing (Load, Stress):** will be performed to monitor the response time of API. QA performance team evaluates **response time, throughput, bottlenecks, and stability** of the API's. These values will be compared with the desired KPI (Key performance indicators) that are agreed as part of test plan
  - **Security testing**: Network, system, client and server side **securities and vulnerabilities of the API** will be tested by the QA security testing team.
- **Smoke, sanity and regression**: These test will be **performed on every new release or code deployments** or new API functionality is deployed. Smoke testing is to make sure code build in test environment is successful, sanity ensures new features are deployed and ready for testing, regression will be performed to make sure already existing API functionalities are not broken. Automating these tests would be best practice
- **UI Testing**: Handful of UI testing need to be performed to make sure **frontend application is able to invoke the rest service** and read response effectively

Question: Which tools would you suggest using to help your team with a daily work?

Test tool selection depends purely on quantity of API development, project budget and resource expertise. But I prefer the below tool implementation for testing Rest API's & and other management tools

- **Test execution**: **Postman** is the best tool for working with rest services. It is light weight and easy to learn. It even has advanced functionalities like designing automation scripts,

integration with Jenkins, etc., **Insomnia** is new in the market and pretty faster than postman, widely used these days. But if my main intention is test automation then I will go with **SOAP UI**, main advantage is, it also accommodate SOAP requests (postman can also do but needs additional plugin to extract request XML). **Rest-Assure** is also best option in case you are looking tool exclusively for Rest services testing (supports BDD)

- **Plugins**: **JSON formatter** browser plugin, Some times you can run the rest services directly on the browser, in case if you have larger amount of response data then it would be difficult to read the response. JSON formatter comes very handy in daily testing activities. **JSON and XML validators** will help in validating the rest requests
- **Defect management tool**: **Jira**, widely used, simple and advanced. Even it can be used as test management tool with **TestFLO plugin**
- **Test management tool**: **HP ALM**, licensed, every flexible and powerful. **Rally** would be best option if we are following agile methodology
- **Team management & test reporting**: **Confluence**, best option for shared work space. Very helpful for cross functional teams. Powerful pages, dashboard creation features, test teams can publish the test metrics smoothly using Confluence pages
- **CI/CD – Jenkins, Git** (version control, if the project is maintained on a shared repository)

Question: If you would do a test automation which techniques or best practices would you use the application?

Automation best practices:

1. Choose an automation tool which is flexible and capable to automate different type of applications (API, Web, mobile, etc.,) Rather then having multiple frameworks for API, Web, the best approach is **build functional testing of application on a single framework/tool**.
2. **Functional and Non functional API test automation need to be designed and maintained as separate frameworks**, having on same framework will make things complex and often creates confusion.
3. **Test cases must be independent** and must include assertion statements
4. Framework **should be maintained using dependency managers**, like maven-POM, docker
5. We need to find the **priority test cases for automation** first and then we need to start automating
6. **Smoke, Sanity and Regression test runs should be executed regularly**. Running these test cases on every release/ Code deployments need to be mandatory
7. **Code reuse wherever possible in the framework.** Best approach to use shared directories within the framework
8. Always we need to follow the best practices defined by the scripting languages we are using.
9. Using wait functions to wait on a condition or specified time period
10. Creating **automation suite for unit testing** and suggesting development team to run them in the post build activity of their code deployment is one of the best approach. This helps developers immediately observe the breakdowns in the code deployments, fix and redeploy.
11. Maintaining **API object model** (similar to POM for UI testing) will be a good approach, in this case you will have one resource file for each API functionality. File includes, URI, request and

response parameters (mandatory and non mandatory), different request actions (post request, read response parameters). In this way any changes in the API can be easily adopted by your automation framework. Makes code update easier

12. Your **API resources file needed to be aligned with API versions deployed**, meaning multiple versions of API files need to be created. This helps automation execution to switch back and forth between different versions of API, whenever needed.
13. **Global variables** like environment details, API authorizations, default timeouts need to be specified **in a common directory** and they should be shared across the API script files
14. **Methods names must be clear and unique**. Instead of writing 'Read response ()', writing 'Read response of operation ()' is best approach
15. Tests should able to **expect and handle system (404, 500, etc.,) and user defined error codes** and messages. Tests should be gracefully completed in case of error in response
16. Both **TDD and BDD are best approaches** for API testing
17. We should have a **JSON/XML requests repository** in the automation framework
18. **One test script file for each API functionality**, each test script file should include all possible tests of API with all valid and invalid request parameters
19. **Maintain tags** (@Sanity, @Regression, @positive, etc.,) this way we can run sanity/smoke/regression test cases easily without creating any additional test suites
20. Automation API request should not wait forever, it need to **wait for a defined response time** and gracefully fail after that time.
21. **Some front end UI test automation** should also need to be included as part of API testing
22. **Every API test case should validate response status code**, Reading response parameters should happen based on response status code.
23. Automation framework should be flexible to integrate with CI/CD tools and also able to run on docker container. **Non-technical QA team members should also be able to run the automation builds**
24. **Generating HTML Test reports**. reports should also include response time of each request
25. **Effective logging**. API testing don't have UI, so we won't able to have test artifacts (screenshots). We need to capture and write the unique reference ID/Response ID, session ID, response content to the Log file. Having DEBUG as log level is the best approach
26. We already have the logic in our functional test cases for REST, so its better use the same calls/logic to do performance testing
27. Generating aggregated test results and asserting response time with expected KPI values is best approach in performance testing
28. Knowledge of automation testing should be shared across the team, anyone from QA team should be able to easily execute the automation builds and understand test execution results