Imagine the following situation: you are joining a cross-functional team which builds a

front-end application using REST APIs. You are a first QA engineer and need to establish a

QA process in the team.

1. What would you do in your first few days of work? Where would you start?

2. Which process would you establish around testing new functionality?

3. Which techniques or best practices in terms of code architecture and test design would you use in your automated tests?

Answers:

1. Drawing on my previous experience as the first QA Engineer, and following the measure-assess-plan model, my first priorities would be the following:

- Attempt to meet every team member in a one-on-one encounter, preferably informal, establishing an initial communication, and understanding their role, how they see themselves in the team, what their best skills are, and what possible insight they might have on areas where the team could improve, especially how the team could make use of a specialized QA Engineer to drive the testing process.
- Get familiar with the current testing activities, no matter how limited their scope and coverage might be. Investing some time measuring what the current practices are will pay off in the long run in case some ideas are re-usable. This can include getting familiar with coverage strategies (if any exist), any automation code, any previously designed tests and their possible documentation, and existing static analysis methods and tools, to name a few things.
- Get as familiar as possible with the product or family of products at hand. This is not just limited to the technology stack in use, although that's definitely a useful detail. This will help me establish the product architecture, design concepts, available interfaces, the type and size of data being processed, transferred and stored.
- Get familiar with the method and depth of requirements documentation, is process modeling used? Is Gherkin used? What could be the best way to directly reflect those requirements in automated tests? Automation friendly approaches to requirements are obviously an ideal scenario, but understanding any existing approaches and how they can be used as well as improved is one of the most important factors in a SDLC, since every other activity will depend on the successful communication of requirements.
- Get familiar with the current deployment strategies, and their level of automation. Do staging environments exist as a standalone part of the pipeline? Do they have their own automated deployment, or is the staging environment maintained manually?
- Evaluate the possible skill gaps (in my own skill set) that might exist, whether this is the result of the technology stack in use, the approach to automation, the nature of the industry, or any other factor.

Those activities should cover the measure and (some of the) assess activities. The planning will depend on gathering as much data and knowledge as possible during this first phase.

2. Establishing a QA process is highly dependent on the SDLC implemented, but assuming an iterative approach, specifically Agile methodologies, I would highlight the below concepts as pillars of the testing approach:

- Early testing: A QA engineer should be involved as early as possible in the software lifecycle, whether participating in requirements reviews, getting familiar with the business use cases for the product in question, or even designing the test and outlining the test automation code skeleton as early as possible. This can increase the effect of communication in the team and help the testing process, and ultimately increase the velocity of the team.
- Continuous testing: The regression suite for a product should be run as often as possible, in order to detect defects almost as soon as they are introduced. I would ideally implement a daily run, or possibly even running the tests twice a day.
- Defining a review checklist from a testing point of view, to be implemented for the review of artifacts, especially requirements.
- Automation by design: The approach to product development, requirements documentation, and test design should take into account test automation as early as possible in the lifecycle. This is also sometimes referred to as design for automation.
- Tool selection for test automation, and other activities that require tool support. The tools should allow the different functions in the team to seamlessly integrate and communicate, for the best available cost. Where budgets are not available for tool support, choosing the open source tools or freeware tools that are easily extendable is the best approach.
- Test planning: Defining the mechanism for prioritization of test cases or features. The role of risk assessment should be clearly defined, and its level of formality agreed upon. Given the definition of this process, we can then perform efficient test planning at early stages of an epic, within a sprint, and even for test cases within a feature.
- Evaluate the feature dependencies, and define in the test documentation what feature or occasionally other test (though this is to be avoided) each test depends on.
- Test monitoring and control: metrics selection, coverage strategy definition.
- Test reporting: A dashboard for test execution data should be made available to a wide variety of stakeholders. Choosing a monitoring approach is one of the most important things to define in a QA process.
- Test strategy documentation: put in writing the design concepts behind all the testing activities. This will help future maintenance of artifacts, especially if a high level of abstraction is used in test automation.
- Building and maintaining a common defects database, to be used in future exploratory testing, and possibly to expand the automated regression suite.

3. This is a strategic matter that could also depend on specific organization and SDLC concern, but in general I could recommend the below things to any organization:

   Test design:

- The key is independent tests, execution order should not matter to the automated testing process, no matter how complex each test case is, or what test level (of the test pyramid) it pertains to. In fact, the randomization of the execution order of tests is a good way of catching additional defects, that executing in the normal order every time might miss.
- Test documentation is of the essence, preferably in an automation friendly way, or at least as one of the ways that tests are documented. I find Gherkin to be an excellent umbrella that marries requirements, test scenarios, and test data, if the Gherkin feature files are written right. This test documentation should include test dependencies (if they must exist), and should also include each test's software module dependency (i.e. what should be running in order for the test to pass, to avoid "false positives" with bug reports).
- Documenting the test data, or at the very least their generation methods (scripts, tools, in-framework classes or functions). This will make the data maintenance easier and allow for repeatable results.
- Clearly defined expected results for each test case. Again, Gherkin is a great solution for this. This is not to say that a good old spreadsheet wouldn't do the job, as long as it's detailed, accurate, and up-to-date.

Test automation code:

- Should be platform independent, and this is not just a general portability concern for the case where the production, staging, or any other deployment platform changes, this is due to the fact that the development workstation's platform is often different from the target environment where the test execution occurs, especially if it's part of a CI/CD pipeline.
- Should be interface independent, by which we mean the actual interface it is testing (API, UI, network protocol like a TCP or UDP server, etc.). This is achieved by separating the adaptation layer, in which the actual driver used to communicate with the interface is instantiated and used. This driver, usually a 3rd party tool, is subject to change, or even to being discontinued. By separating the test adaptation layer, we mitigate this risk and lower the maintenance cost of our framework.
- Should allow for the definition of business-related keywords, or at least define some sort of abstraction layer for readability. A test automation framework should be viewed, reviewed, and referred to by everyone in a cross-functional team, if not also used by everyone. Ideally, a test automation framework should allow even non-technical staff like business analysts to define some smoke tests and view how the software under test behaves in those tests.
- Should have enough layers and detailed enough reporting to show both the high level test steps and the exact low level action that may have caused a test to fail.
- Should store away UI selectors and other low level interaction data in relevant objects. A lot of test automation engineers opt for an application architecture that follows the page object model. In my personal opinion, while the page object model is a great template for the majority

of situations, a test automation engineer should be comfortable deviating from it and keeping some flexibility in the test automation framework design, especially if the framework is to support more than one product, and more than the UI of the product.

- Should be extremely well documented, showing users from different types of audiences the details and intended usage of all the features in it.
- Should separate the test data from the automation code, and have tools to read data from popular formats like CSV and JSON, but also some utilities to generate test data.
- Automation logic should reflect the requirements, this way the change in automation code reflects the change in the product, without being sensitive to the particularities of the interfaces or trivial changes in UI design. The maintenance effort for the test automation code should not exceed the scale of the maintenance effort put into the software under test itself.
- Should take into consideration isolating the repetitive preconditions into suite setup, test setup, suite teardown, and test teardown sections.
- Should ideally allow the execution of the test suite as part of a CI/CD pipeline, separately, in a distributed fashion, and support parallel processing. One or more of these features can be useful at any time.
- Should support test selection using tags or any other identifiers, as we don't necessarily always want to use the framework to run the full regression suite.
- Should employ naming conventions, ideally the same ones used in the test analysis and design, and also in relation to the product.
- Should not ignore security, and should provide some features to securely transfer and handle test data.