Advanced Principles of API Testing | Part 1

Presented by:

Varuna Srivastava

ThoughtWorks Canada

Brought to you by:
Varuna Srivastava

Varuna is a technical tester who's worked on award-winning projects across a wide variety of technology sectors, including retail, travel, financial, and the public sector, and worked with various web, mobile, and IoT technologies. Varuna is a passionate advocate of shipping quality code to production using agile practices. When not working, Varuna likes to get her hands dirty experimenting with her culinary skills. Most of her weekends are spent in cookgraphy—cooking plus photography!
Advanced Principles of API Testing
About Me

- Lead Quality Analyst at Thoughtworks
- Have been in Test automation for 10 yrs
- Testing traveler
- Conference Organizer @EuroTestConf
- Twitter handle: @vibranttester
Agenda

- Role of API’s
- API architecture
- What is API testing
- Types of api testing
- API documentation tools
- API design patterns
- API Test automation
Web services

Response from server to client

Request from client to server

Server hosting the web service
Web services

@vibranttester
REST architecture

Level 3: Hypermedia Controls

Level 2: HTTP Verbs

Level 1: Resources

Level 0: The Swamp of POX

Glory of REST

@vibranttester
API architecture

- **Http verbs**
- **Http headers**
- **Http response**
- **Http body**
Http verbs

POST
PUT
DELETE
GET
Http headers

- Content Type
- Cache Control
- Authorization
- Set Cookie
Http headers → Set headers

given().request()
.with()
.contentType("application/json")
.header( headerName: "auth-token", basicAuthToken)
.header( headerName: "set-cookie", sessionID)
.header( headerName: "cache-control",
          headerValue: "max-age:604800")
Http Response

200 OK

201 Created

304 Not Modified

400 Bad Request

401 Unauthorized

404 Not Found
Rest API structure

given().request()
  .with()
  .contentType("application/json")
  .header(headerName: "auth-token", basicAuthToken)
  .header(headerName: "set-cookie", sessionId)
  .header(headerName: "cache-control", headerValue: "max-age:604800")
  .QueryParam(parameterName: "format", ...parameterValues: "json")
  .body(new CreateAddressRequestBuilder().build())
  .when()
  .post(path: "http://localhost:8080/addresses")
  .then()
  .assertThat()
  .statusCode(201)
  .body(path: "addressId", notNullValue());
API Testing
Test Pyramid

@vibranttester
API Testing (Adapted from watirmelon blog)

Manual Tests

Automated GUI Tests

Integration Tests

Unit Tests

Ideal Software Testing Pyramid

Manual Session Based Testing

Automated GUI Tests

Automated API Tests

Automated Integration Tests

Automated Component Tests

Automated Unit Tests

Software Testing Ice-Cream Cone Anti-Pattern

@vibranttester
Types Of API Testing
Types of API Testing

- Functional
- Security
- Load
- Contract
- Runtime error
- Validation
API Functional Tests

- Focus on testing the functionality of respective API with valid inputs
  /searchItem By name By brand

Responsibility:
- Define scope of API
- Verify edge case scenario
- Verify handled error scenario
API Contract Tests

- Focus on the messages that flow between a consumer and provider /orders

Responsibility:
- bugs in the consumer
- misunderstanding from the consumer about end-points or payload
- breaking changes by the provider on end-points or payload
API Load Tests

- Focus on verifying whether the theoretical solution works as a practical solution under a given load.

Responsibility:
- Verify how scalable APIs are at maximum user load
- Verify how quickly APIs respond i.e. speed
- Verify if the APIs are stable under varying loads
API Security Tests

- Focus is to make your data safe from hackers, and ensure that the API is as safe as possible

Responsibility:
- Validated external threats
- Fuzz Testing
- Penetration testing
API Documentation
### Swagger

**pet**  
Everything about your Pets

<table>
<thead>
<tr>
<th>Method</th>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>POST</strong></td>
<td>/pet</td>
<td>Add a new pet to the store</td>
</tr>
<tr>
<td><strong>PUT</strong></td>
<td>/pet</td>
<td>Update an existing pet</td>
</tr>
<tr>
<td><strong>GET</strong></td>
<td>/pet/findByStatus</td>
<td>Finds Pets by status</td>
</tr>
<tr>
<td><strong>GET</strong></td>
<td>/pet/findByTags</td>
<td>Finds Pets by tags</td>
</tr>
<tr>
<td><strong>GET</strong></td>
<td>/pet/{petId}</td>
<td>Find pet by ID</td>
</tr>
<tr>
<td><strong>POST</strong></td>
<td>/pet/{petId}</td>
<td>Updates a pet in the store with form data</td>
</tr>
<tr>
<td><strong>DELETE</strong></td>
<td>/pet/{petId}</td>
<td>Deletes a pet</td>
</tr>
<tr>
<td><strong>POST</strong></td>
<td>/pet/{petId}/uploadImage</td>
<td>uploads an image</td>
</tr>
</tbody>
</table>

**store**  
Access to Petstore orders

<table>
<thead>
<tr>
<th>Method</th>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GET</strong></td>
<td>/store/inventory</td>
<td>Returns pet inventories by status</td>
</tr>
<tr>
<td><strong>POST</strong></td>
<td>/store/order</td>
<td>Place an order for a pet</td>
</tr>
</tbody>
</table>
API Blueprint

FORMAT: 1A
# Dredd example
## Addresses [/addresses]
### Create Address [POST]
+ Request (application/json)
  ```
  "addressId": "1",
  "title": "Mrs",
  "firstName": "Varuna",
  "lastName": "Srivastava",
  "line1": "300 Front St West",
  "line2": "Blue building",
  "line3": "Box",
  "city": "Toronto",
  "state": "Ontario"
  ```
+ Response 201 (application/json; charset=utf-8)
Dredd commands

`npm install -g dredd`

dredd init

dredd
Dredd result

POST /addresses

Create Address

POST /addresses

Create Address

https://app.apiary.io/public/tests/run/a017f6eb-e89e-4afe-8d49-327559c08d24

@vibranttester
API Design Patterns
Design Patterns
Why Design Patterns in Test Automation?

- Reliability
- Scalability
- Flexibility
- Reusability
- Maintainability

@vibranttester
Types of Design Patterns

- Creational
- Behavioural
- Structural

@vibranttester
Structural Design Patterns
Structural Design Pattern

Structural Design Patterns are used to avoid duplicates in code and increase the readability and navigation of code in project.
Page Object Pattern

HomePage

Search Result Page

Search Item

Item Search Tests
Page Object Pattern

SearchItemTests.java

Tests and method invocations

SearchItemPage.java

Method assertions

SearchItemIds.java

Identifier of an element
Page Object Pattern

```java
@Test(groups = Categories.SANITY)
public void verifySearchResults() throws InterruptedException {
    JourneyDetails journeyDetails = new JourneyDetailsBuilder().build();
    searchResultsPage = homePage.searchForAOneWayJourneyWith(journeyDetails);
    searchResultsPage
        .verifyCheapestIsSelected()
        .verifySearchResultsAreSortedByPrice();
}
```
Page Factory Pattern encapsulates page’s attribute by `findby` annotations. It helps to work directly with page fields hiding the low level complexity.
```java
public class HomePage {
    WebDriver driver;
    SearchResultsPage searchResultsPage;
    private By searchTextBox=By.id("twotabsearchtextbox");
    private By submitText=By.className("nav-input");

    public HomePage(WebDriver driver) { this.driver = driver; }
    public SearchResultsPage searchItem() {
        driver.findElement(searchTextBox).sendKeys("...
    }
    return searchResultsPage=new SearchResultsPage(driver);
}
```
public class HomePage {
    WebDriver driver;
    SearchResultsPage searchResultsPage;

    @FindBy(id = "twotabsearchtextbox")
    private WebElement searchTextBox;

    @FindBy(className = "nav-input")
    private WebElement submitText;

    public HomePage(WebDriver driver) {
        this.driver = driver;
    }

    public SearchResultsPage searchItem() {
        searchTextBox.sendKeys(...keysToSend: "Kindle");
        submitText.click();
        return searchResultsPage = new SearchResultsPage(driver);
    }
}
private void launchApplicationUnderTest() {
    PropertyReader reader = new PropertyReader();
    String applicationURL = reader.readProperty("applicationURL");
    driver.get(applicationURL);
    SearchResultsPage searchResultsPage = new HomePage(driver).searchForTheJourney();
}

private void launchApplicationUnderTest() {
    PropertyReader reader = new PropertyReader();
    String applicationURL = reader.readProperty("applicationURL");
    driver.get(applicationURL);
    HomePage homePage = PageFactory.initElements(driver, HomePage.class);
}
Chain of invocation helps to avoid repeating **object** again and again before method invocations and makes code pretty!!
Chain of Invocation Pattern

```java
ReviewOrderResponse roResponse = reviewOrder();
roResponse.assertShippingAddress(addressId, shippingAddress);
roResponse.assertBillingAddresss(shippingAddress);
roResponse.assertPaymentMethod(piId, card, viewOrder().getGrandTotalAmount());
roResponse.assertRootLevelAttributes(viewOrder(), userType: "G");
```

```java
ReviewOrderResponse roResponse = reviewOrder();
roResponse.assertShippingAddress(addressId, shippingAddress)
    .assertBillingAddresss(shippingAddress)
    .assertPaymentMethod(piId, card, viewOrder().getGrandTotalAmount())
    .assertRootLevelAttributes(viewOrder(), userType: "G");
```
◊Chain of Invocation Pattern

```plaintext
addItemToCart() AddItemsToCartResponse
    .addShippingAddress() AddShippingAddressResponse
    .addPaymentInstructionWithBillingAddress() AddPIResponse
    .submitOrder() OrderSubmitResponse
    .assertAttributes(orderId);
```
Chain of Invocation Pattern

```java
addItemToCart()
  .addShippingAddress()
  .submitOrder()
  .addPaymentInstructionWithBillingAddress()
  .assertAttributes(orderId);
```
Strategy Design Pattern

Strategy pattern is used whenever we want to have more than one implementations of the same action differently. It makes code more flexible and maintainable by using separation of concepts.
Data Design Patterns
Data Design Pattern

Data Design Patterns are used to separate data and test logic. It reduces amount of data related code from test class.
Value Object makes code more readable and it reduces amount of repeatable constructions. It is immutable which avoid modifications and extensions.
Builder Pattern

Builder pattern makes process of building complex object easier. We don’t have to create multiple constructor for different scenario.
DataProvider Pattern

DataProvider pattern used to provide parameters to a test. A test method will be executed using the same instance of the test class to which the test method belongs.
Creational Design Patterns
Singleton Pattern

Singleton class has only one instance, which provides a global access point to this instance. Singleton object is initialized only when it’s requested for the first time.
◊ Singleton Pattern

Example?
Singleton Pattern

```java
synchronized static SingletonClass getInstance() {
    if (instance == null)
        instance = new SingletonClass();
    return instance;
}

String getAddress() { return "Address of star canada"; }
```
Let’s try this out !!
API Test Automation
Comments..?Doubts..?Complains..?

Drop a note @vibranttester to continue this conversation

Varuna Srivastava
LEAD QUALITY ANALYST
varunas@thoughtworks.com  |  thoughtworks.com