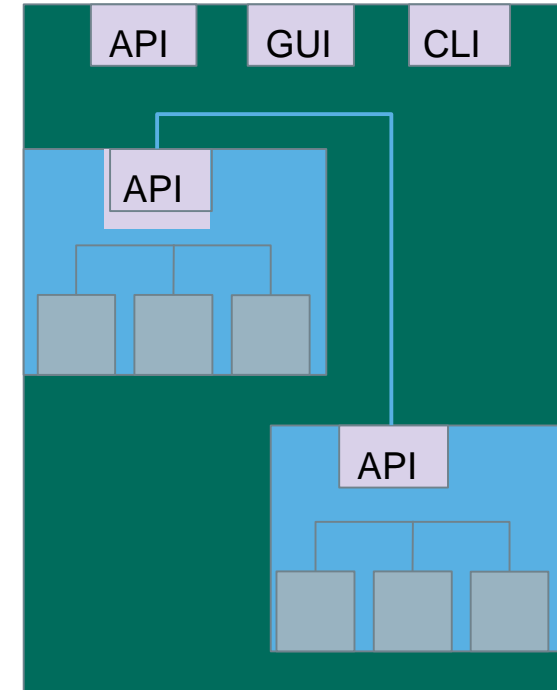# System Testing

CSE 4495- Lecture 4 - 05/07/2022

Instructor : Md. Mohaiminul Islam

Testing Stages

- We interact with **systems** through **interfaces**.
  - APIs, GUIs, CLIs
- Systems built from **subsystems**.
  - With their own interfaces.
- Subsystems built from **units**.

    - Communication via method calls.
    - Set of methods is an interface.

Testing Stages

- **Unit Testing**
  - Do the methods of a class work?
- **System-level Testing**
  - **System (Integration) Testing**

    - (Subsystem-level) Do the collected units work?
    - (System-level) Does high-level interaction through APIs/UIs work?

  - **Exploratory Testing**
    - Does interaction through GUIs work?
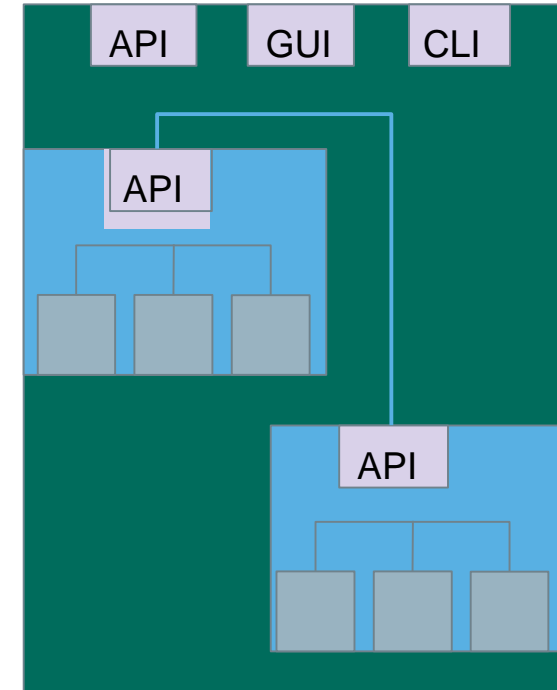
Lecture Goals

- Discuss testing at the system level.
  - System (Integration) Testing versus Unit Testing.
  - Introduce process for creating System Tests.
  - Identify a Independently Testable Function
  - Identify Choices
  - Identify Representative Values
  - Generate Test Case Specifications
  - Generate Concrete Test Cases

System Testing

- Tests the **integration** of units as a whole.
    - Integrate units in one subsystem.
    - Then integrate the subsystems.
- Test through a **defined interface**.
    - Focus on showing that functionality accessed through interfaces is correct.
    - Subsystems: "Top-Level" Class, API
    - System: API, GUI, CLI, …

Unit vs System Testing

- Unit tests focus on a **single class**.
  - Simple functionality, more freedom.
  - Few method calls.

- System tests **bring many classes together**.
  - Focus on testing through an interface.
  - One interface call triggers many internal calls.
    - Slower test execution.
  - May have complex input and setup.

System Testing and Requirements

- **Tests can be written early in the project.**
    - Requirements discuss high-level functionality.
    - Can create tests using the requirements.
    - **System testing does not require a detailed design.**
- Creating tests supports requirement refinement.
- Tests can be made concrete once code is built.

Interface Types

- ## Parameter Interfaces
  - Data passed from through method parameters.
  - Subsystem may have interface class that calls into underlying classes.

- ## Procedural Interfaces
  - Interface surfaces a set of functions that can be called by other components or users (API, CLI, GUI).
  - Integrates lower-level components and controls access.
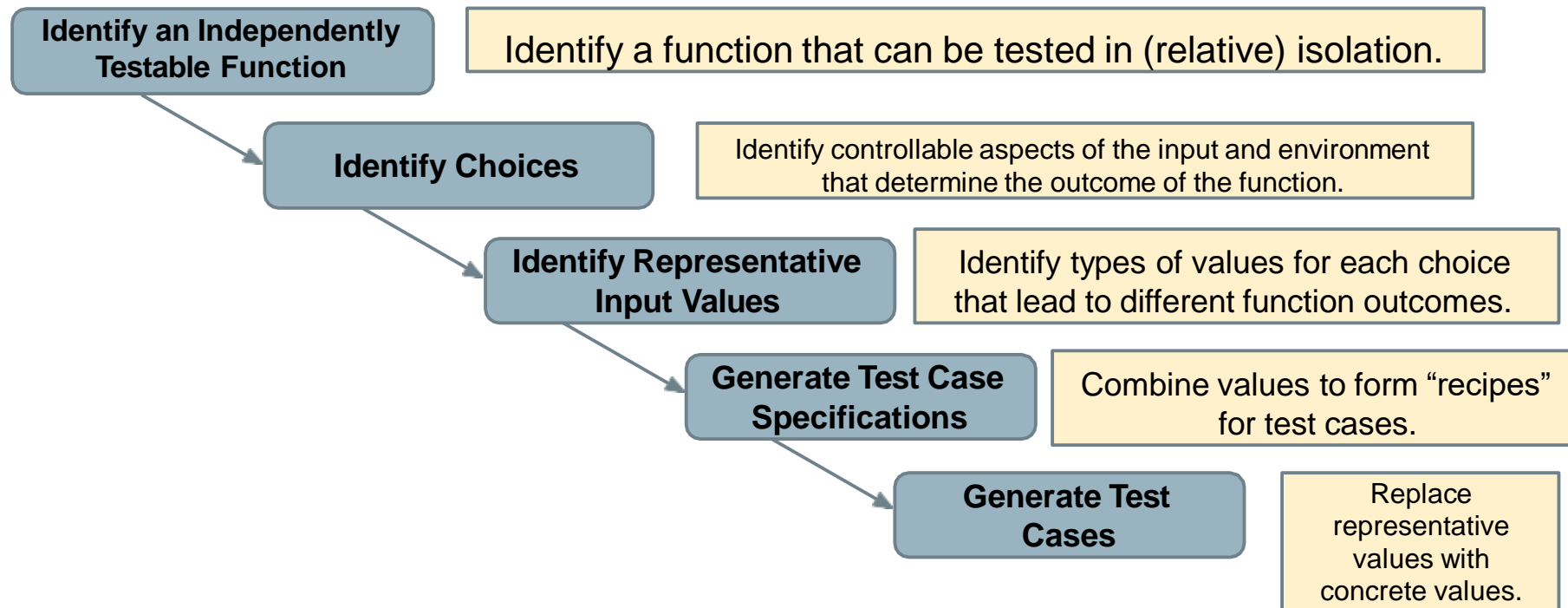
Interface Types

- Shared Memory Interfaces
  - A block of memory is shared between (sub)systems.
    - Data placed by one (sub)system and retrieved by another.
  - Common if system architected around data repository.

- Message-Passing Interfaces
  - One (sub)system requests a service by passing a message to another.
    - A return message indicates the results.
  - Common in parallel systems, client-server systems.

Interface Errors

- Interface Misuse
  - Malformed data, order, number of parameters.

- Interface Misunderstanding
  - Incorrect assumptions made about called component.
  - A binary search called with an unordered array.

- Timing Errors
  - Producer of data and consumer of data access data in the wrong order.

Creating System-Level Test Cases

Creating System-Level Tests

Identify an Independently Testable Function

Identify a function that can be tested in (relative) isolation.

Identify Choices

Identify controllable aspects of the input and environment that determine the outcome of the function.

Identify Representative Input Values

Identify types of values for each choice that lead to different function outcomes.

Generate Test Case Specifications

Combine values to form "recipes" for test cases.

Generate Test Cases

Replace representative values with concrete values.

Independently Testable Functionality

- **A well-defined function that can be tested in (relative) isolation.**
  - Based on the "verbs" - what can we do with this system?
  - The high-level functionality offered by an interface.
  - UI - look for user-visible functions.
    - Web Forum: Sorted user list can be accessed.
    - Accessing the list **is** a testable functionality.
    - Sorting the list is **not** (low-level, unit testing target)

13

Units and "Functionality"

- Many tests written in terms of "units" of code.
- An independently testable function is a *capability* of the software.
  - Can be at class, subsystem, or system level.
  - **Defined by an interface.**

Identify Input Choices

- What choices do we make when using a function?
  - **Anything we control that can change the outcome.**
- What are the *inputs* to that feature?
- What *configuration choices* can we make?
- Are there *environmental factors* we can vary?
  - Networking environment, file existence, file content, database connection, database contents, disk utilization, …

Ex: Register for Website

- What are the inputs to that feature?
  - `(first name, last name, date of birth, e-mail)`
- Website is part of product line with different database options.
  - `(database type)`
- Consider implicit environmental factors.
  - `(database connection, user already in database)`

16

Parameter Characteristics

- Identify choices by understanding how parameters are used by the function.

- Type information is helpful.
  - `firstName` is string, database contains `UserRecords`.

- … but context is important.
  - Reject registration if in database.
  - … or database is full.
  - … or database connection down.

Parameter Context

- Input parameter split into multiple "choices" based on contextual use.
  - "Database" is an implicit input for User Registration, but it leads to **more than one** choice.
  - "Database Connection Status", "User Record in Database", "Percent of Database Filled" influence function outcome.
    - **The Database "input" results in three input choices when we design test cases.**

Examples

Class Registration System

What are some independently testable functions?

- Register for class
- Drop class
- Transfer credits from another university
- Apply for degree

Example - Register for a Class

What are the choices we make when we design a test case?

- Course number to add
- Student record
- What about a course database? Student record database?
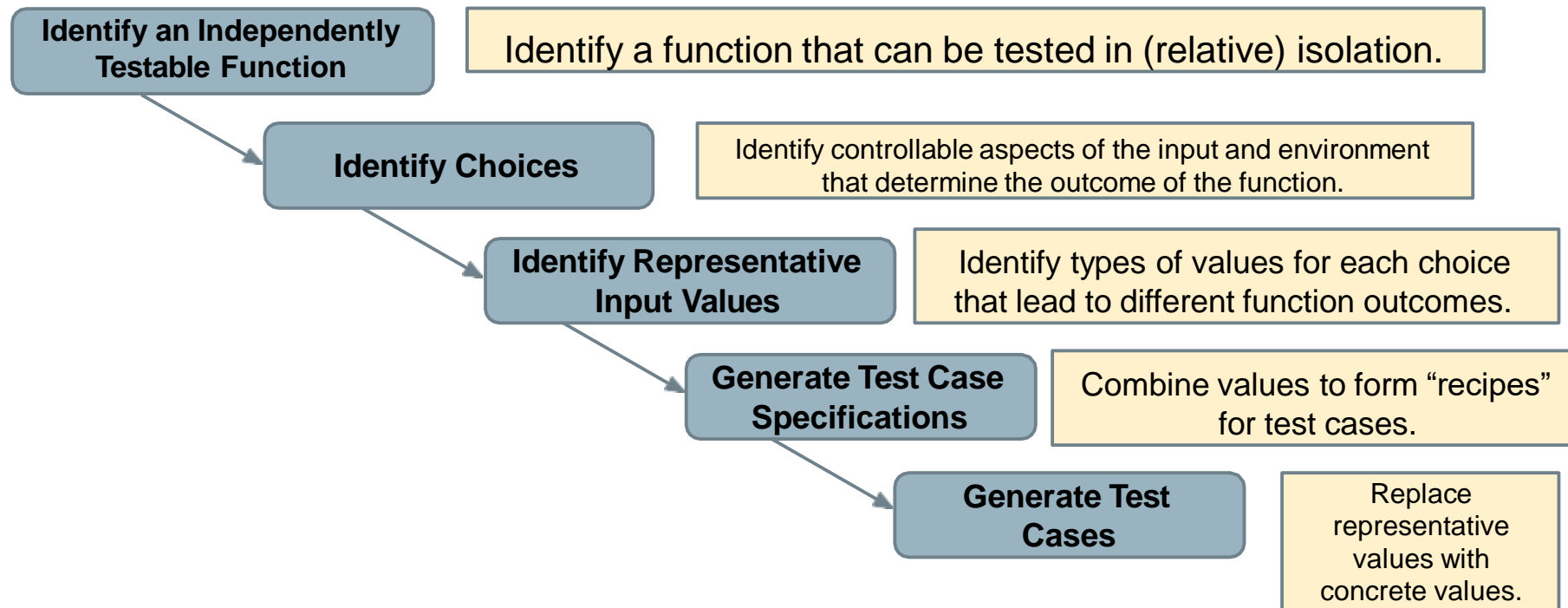- **What else influences the outcome?**

Example - Register for a Class

- Student Record is an implicit input choice.

- How is it used?
    - Have you already taken the course?
    - Do you meet the prerequisites?
    - What university are you registered at?
    - Can you take classes at the university the course is offered at?

Example - Register for a Class

- Potential Test Choices:
  - Course to Add
  - Does course exist?
  - Does student record exist?
  - Has student taken the course?
  - Which university is student registered at?
  - Is course at a valid university for the student?
  - Can student record be retrieved from database?
 - Does the course exist?
  - Does student meet the prerequisites?

Creating System-Level Tests

Identify an Independently Testable Function

Identify a function that can be tested in (relative) isolation.

Identify Choices

Identify controllable aspects of the input and environment that determine the outcome of the function.

Identify Representative Input Values

Identify types of values for each choice that lead to different function outcomes.

Generate Test Case Specifications

Combine values to form "recipes" for test cases.

Generate Test Cases

Replace representative values with concrete values.

Identifying Representative Values

- We know the functions.
- We have a set of choices.
- What values should we try?
  - For some choices, finite set.
  - For many, near-infinite set.

- **What about exhaustively trying all options?**

Test Input Data

Program

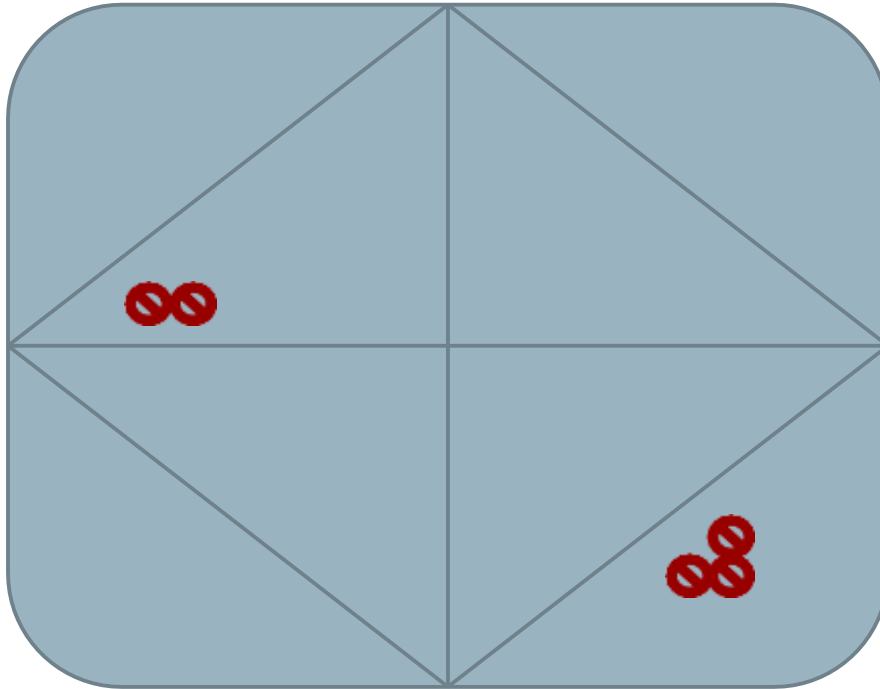Test Output Results
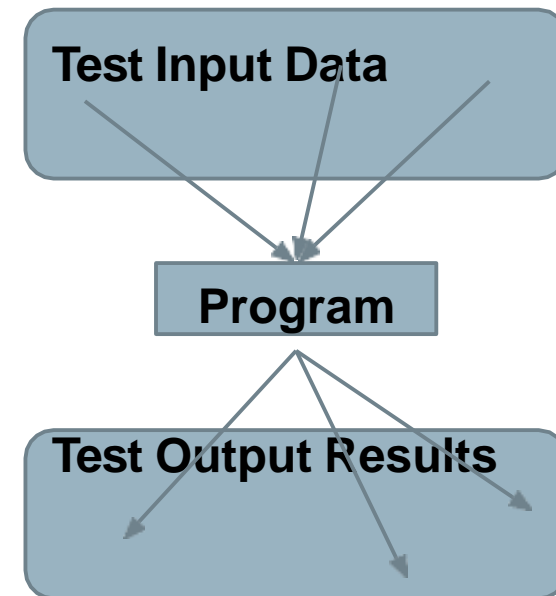
Not all Inputs are Created Equal

- Many inputs lead to same outcome.
- Some inputs better at revealing faults.
  - We can't know which in advance.
  - Tests with different input better than tests with similar input.

Test Input Data I

Program

Test Output Results O
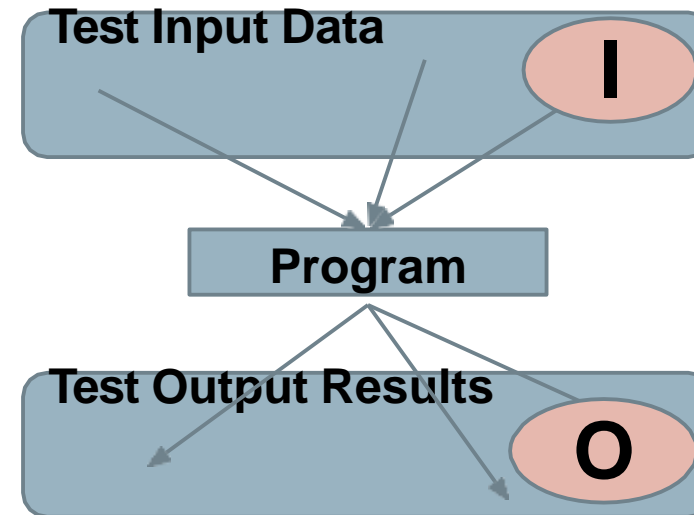
Input Partitioning

- Consider possible values for a variable.

- Faults sparse in space of all inputs, but dense in parts where they appear.
  - Similar input to failing input also likely to fail.

- Try input from partitions, hit dense fault space.

Equivalence Class

- Divide the input domain into **equivalence classes**.
  - Inputs from a group interchangeable (trigger same outcome, result in the same behavior, etc.).
  - If one input reveals a fault, others in this class (probably) will too. In one input does not reveal a fault, the other ones (probably) will not either.

- Partitioning based on intuition, experience, and common sense.

Example

```
substr(string str, int index)
```

What are some possible partitions?

- index < 0
- index = 0
- index > 0
- str with length < index
- str with length = index
- str with length > index
- ...

Choosing Input Partitions

- Equivalent output events.
- Ranges of numbers or values.
- Membership in a logical group.
- Time-dependent equivalence classes.
- Equivalent operating environments.
- Data structures.
- Partition boundary conditions.

Look for Equivalent Outcomes

- Look at the outcomes and group input by the outcomes they trigger.
- Example: **getEmployeeStatus(employeeID)**
  - Outcomes include: Manager, Developer, Marketer, Lawyer, Employee Does Not Exist, Malformed ID
  - Abstract values for choice `employeeID`.
    - Can potentially break down further.

30

Look for Ranges of Values

- Divide based on data type and how variable used.
  - Ex: Integer input. Intended to be 5-digit:
    - < 10000, 10000-99999, >= 100000
    - Other options: < 0, 0, max int
    - Can you pass it something non-numeric? Null pointer?
- Try "expected" values and potential error cases.

31

Look for Membership in a Group

Consider the following inputs to a program:

- A floor layout

- A country name.

- All can be partitioned into groups.
  - Apartment vs Business, Europe vs Asia, etc.

- Many groups can be subdivided further.

- Look for context that an input is used in.

Timing Partitions

- Timing and duration of an input may be as important as the value.
  - Timing often implicit input.
    - Trigger an electrical pulse 5ms before a deadline, 1ms before the deadline, exactly at the deadline, and 1ms after the deadline.
    - Close program before, during, and after the program is writing to (or reading from) a disc.

Operating Environments

- Environment may affect behavior of the program.
- Environmental factors can be partitioned.
  - Memory may affect the program.
  - Processor speed and architecture.
  - Client-Server Environment
    - No clients, some clients, many clients
    - Network latency
    - Communication protocols (SSH vs HTTPS)

Data Structures

- Data structures are prone to certain types of errors.

- For arrays or lists:
  - Only a single value.
  - Different sizes and number filled.
  - Order of elements: access first, middle, and last elements.

ACCESSING ARRAY ITEM

array name (variable)   index

balance[1]

output = 200

By entering the index number of an element, you can get its value. For example, in this case, value of 200 is located at index 1, fetched via syntax of balance[1]

© guru99

balance = {   300   200   100   }
                0   1   2

35
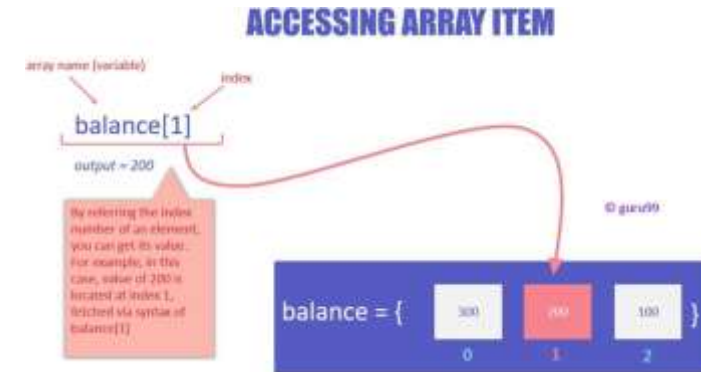
Input Partition Example

What are the input partitions for:

```
max(int a, int b) returns (int c)
```

We could consider `a` or `b` in isolation:

```
a < 0, a = 0, a > 0
```

Consider combinations of `a` and `b` that change outcome:

```
a > b, a < b, a = b
```

Revisit the Roadmap

Identify Representative Values

Generate Test Case Specifications

Generate Test Cases
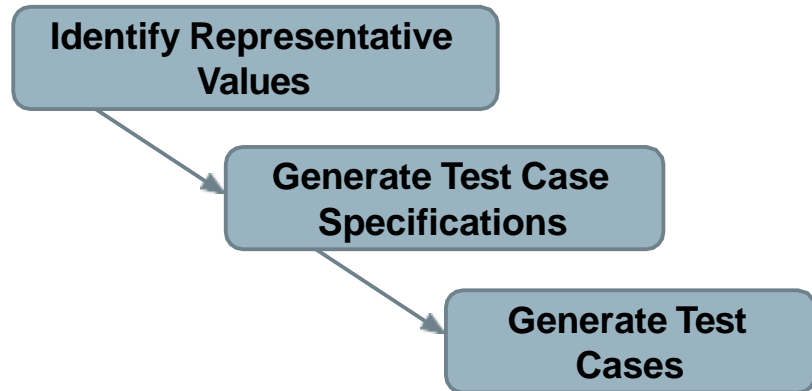
For each testing choice for a function, we want to:
1. Partition each choice into representative values.
2. Choose a value for each choice to form a test specification.
3. Assigning concrete values from each partition.

Forming Specification

Function `insertPostalCode(int N, list A)`.

- **Choice:** int N
  - 5-digit integer between 10000 and 99999
  - **Representative Values:** <10000, 10000-99999, >100000

- **Choice:** list A
  - list of length 1-10
  - **Representative Values:** Empty List, List of Length 1, List Length 2-10, List of Length > 10

Forming Specifications

Choose concrete values for each combination of input partitions:

`insertPostalCode``(int N, list A)`

`int N`

| < 10000 |
| 10000 - 99999 |
| > 99999 |

`list A`

| Empty List |
| List[1] |
| List[2-10] |
| List[>10] |

Test Specifications: (3 * 4 = 12 abstract specifications)

`insert(< 10000, Empty List)`
`insert(10000 - 99999, list[1])`

`insert(> 99999, list[2-10])`

`...`

Concrete Test Cases: (Each specification = 1000s of potential test cases)

`insert(5000, {})`
`insert(96521, {11123})`
`insert(150000, {11123, 98765})`
`...`

Generate Test Cases

Generate Test Case Specifications

Generate Test Cases
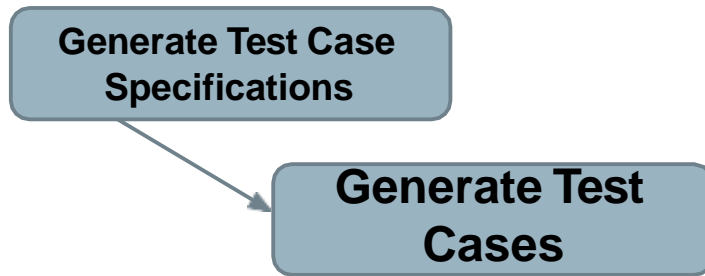
`substr(string str, int index)`

Specification:
`str`: length >=2, contains special characters
`index`: value > 0

Test Case:
`str` = "ABCC!\n\t7"
`index`= 5

Boundary Values

- Errors tend to occur at the boundary of a partition.
- Remember to select inputs from those boundaries.
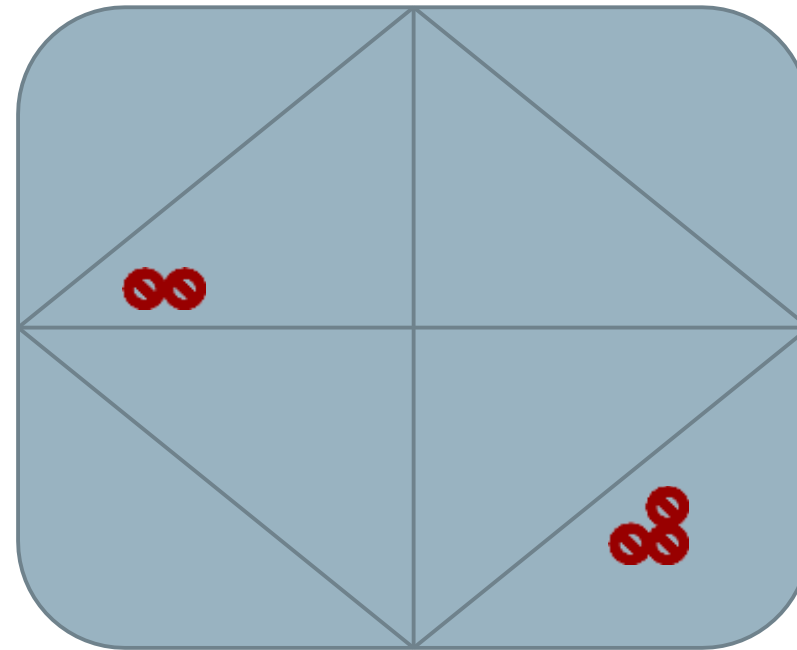
Boundary Values

Choose test case values at the boundary (and typical) values for each partition.

- If an input is intended to be a 5-digit integer between 10000 and 99999, you want partitions:

<10000, 10000-99999, >100000

0		5000	9999					100000	150000	max int

			10000	50000	99999		
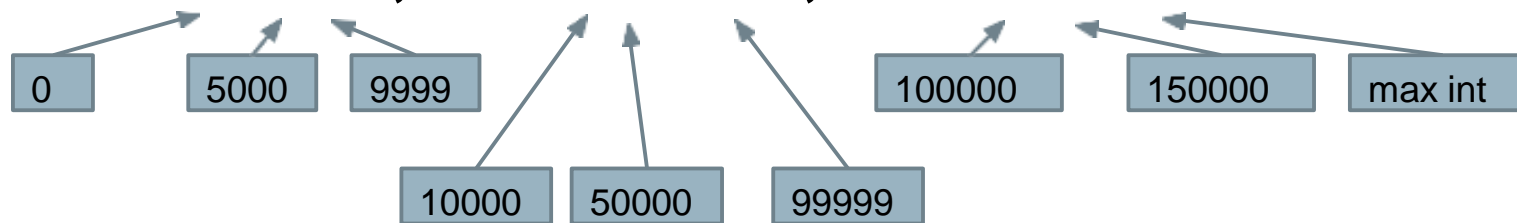
Example - Set Microservice

- Microservice related to Sets:
  - `void insert(Set set, Object obj)`
  - `Boolean find(Set set, Object obj)`
  - `void delete(Set set, Object obj)`
- For each **function**, identify **choices**.
- For each choice, identify **representative values**.
- Create **test specifications** with expected outcomes.

Example - Set Microservice

`void insert(Set set, Object obj)`

Identify an Independently Testable Function

- What are our choices?

Identify Choices

- **Parameter:** set
  - **Choice 1:** Number of items in the set
- **Parameter:** obj
  - **Choice 2:** Is obj already in the set?
  - **Choice 3:** Type of obj (e.g., valid, invalid, null)

Example - Set Microservice

```
void insert(Set set, Object obj)
```

Parameter: obj

Parameter: set

- **Choice:** Number of items in the set
    - **Representative Values:**
        - Empty Set
        - Set with 1 item
        - Set with 10 items
        - Set with 10000 items

- **Choice:** Is obj already in the set?
    - **Representative Values:**
        - obj already in set
        - obj not in set

- **Choice:** Type of obj
    - **Representative Values:**
        - Valid obj
        - Null obj

Example – Set Microservice

Generate Test Case Specifications

Set Size	Obj in Set	Obj Status	Outcome
Empty	No	Valid	Obj added to Set
Empty	No	Null	Error or no change
1 item	Yes	Valid	Error or no change
1 item	No	Valid	Obj added to Set
1 item	No	Null	Error or no Change
10 items	Yes	Valid	Error or no change
10 items	No	Valid	Obj added to Set
10 items	No	Null	Error or no Change
10000	Yes	Valid	Error or no change (may be slowdown)
10000	No	Valid	Obj added to Set(may be slowdown)
10000	No	Null	Error or no Change (may be slowdown)

```
void insert(Set set,
Object obj)
```

- (4 * 2 * 2) = 16 specifications
  - Some are invalid (null in set). Can remove/ignore those.
- Each can become 1+ test cases.

Generate Test Cases

- (1 item, Yes, Valid) becomes:
- **insert({"Bob"}, "Bob");**

46

Creating System Tests for a REST API with Postman

Postman

- Testing and development framework for systems with a REST API.
    - A system interface with **endpoints** we can interact with.
    - At an endpoint, we can send HTTPS request to:
        - **GET** information that you are interested in.
        - **DELETE** the information stored.
        - **PUT** information into what is stored (ex: create a new entry)
        - **POST** information (ex: update an existing record)
- Create requests and test cases using Postman.

Writing Tests in Postman

Test Input
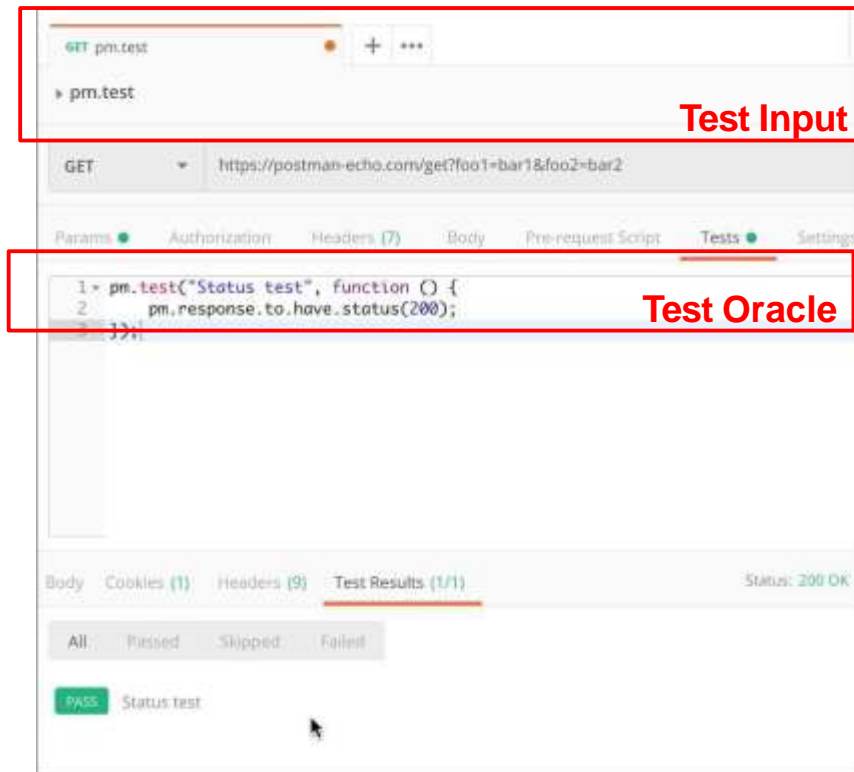
Test Oracle

- Each tab is a request.
- The request is the **test input**.
  - (GET/POST/PUT/DELETE) to an endpoint.
  - Can specify body, header, authorization, etc. for the request.
- Tests tab allows creation of **test oracles**.
  - Write small JavaScript methods to check correctness of output.

1. Select GET as the request type.
2. Set the endpoint URL.
3. Click "Send"
4. The response status is indicated.
5. The body contains the returned information.
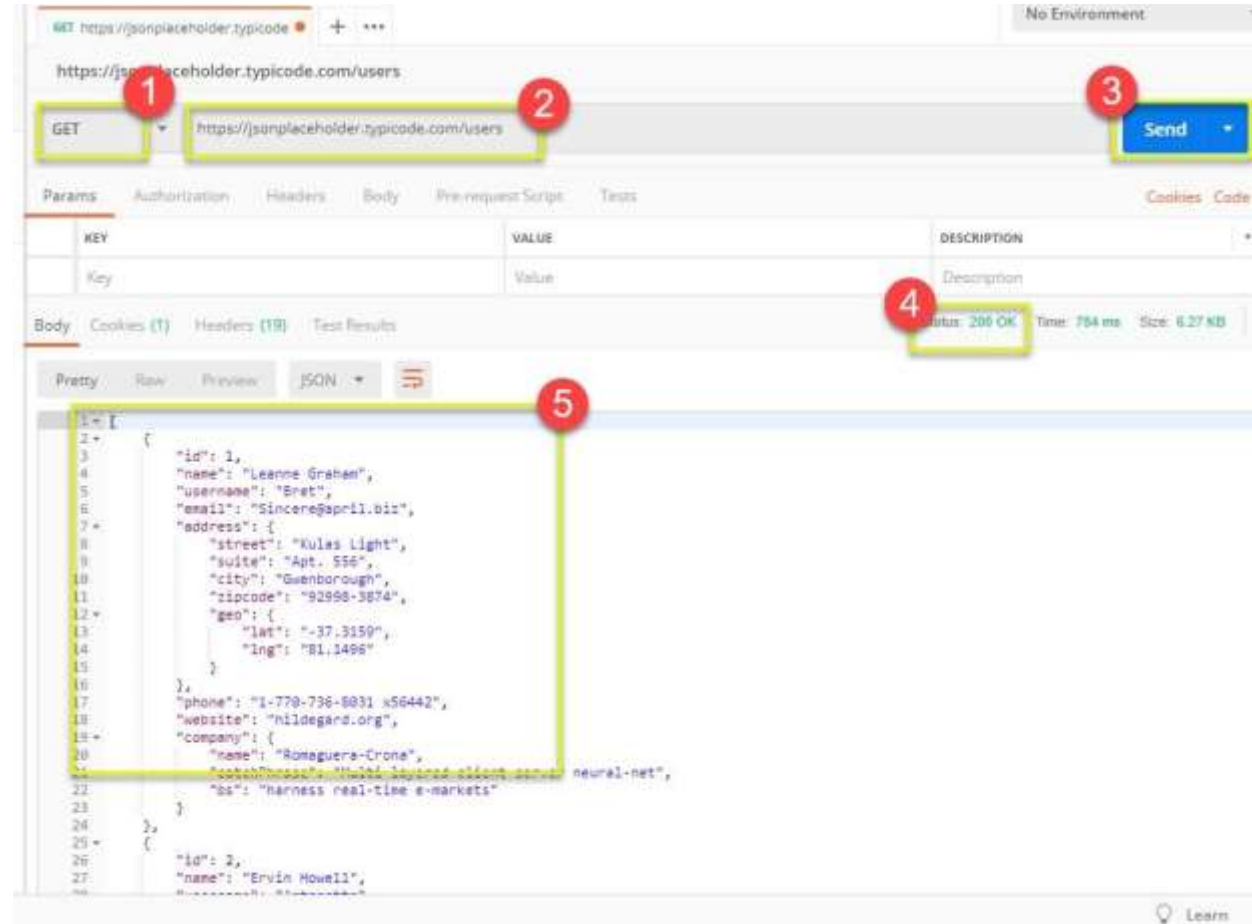
Input - POST

1. Set request to POST.
2. Set the endpoint URL.
3. Select the "Body" tab.

1. Click "raw" (raw text), "binary" (file/executable), etc.
2. Select data format (JSON, XML, etc.)
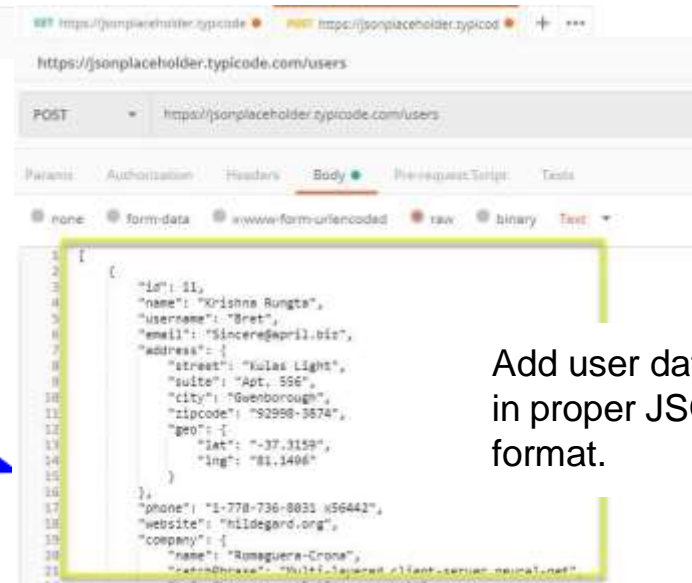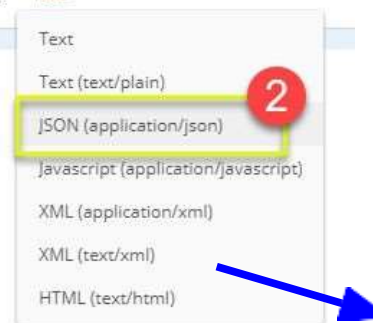
Add user data in proper JSON format.

https://www.guru99.com/postman-tutorial.html

Output - POST

1. Click Send to send request.
2. Response status is indicated (201, data created)
3. Body indicates record "11" was created.

https://www.guru99.com/postman-tutorial.html
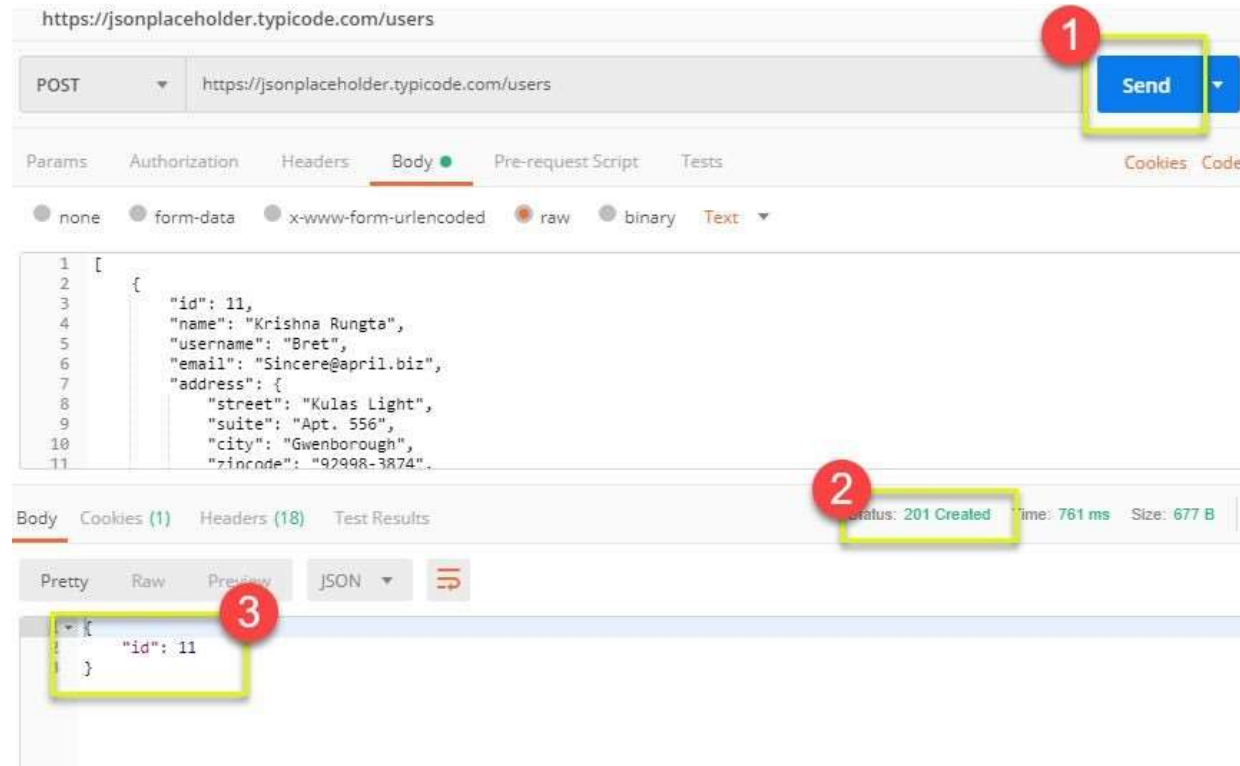
Creating Test Oracles

- Tests tab allows creation of JavaScript blocks used to verify results.
  - These are "test oracles".
  - Embed expectations on results and code to compare expected and actual values.
- pm.test library gives variety of commands to make assertions on output.
  - https://learning.postman.com/docs/writing-scripts/script-references/test-examples/ (many example scripts!)

Oracle Example - Status Check

1. Create test in "tests tab"
2. Snippets offer pre-built test oracles.
3. Example - "status code must be 200"

https://www.guru99.com/postman-tutorial.html

Oracle Example - Expected Value

- Choose snippet "JSON value check"

- This inserts generic test body.

- Change **test name**, **variable to check** (name of the first user), **value to check** (check for name "Leanne Graham").


```
pm.test("Check if user with id1 is Leanne
Graham", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData[0].name).to.eql("Leanne
Graham");
});
```

https://www.guru99.com/postman-tutorial.html
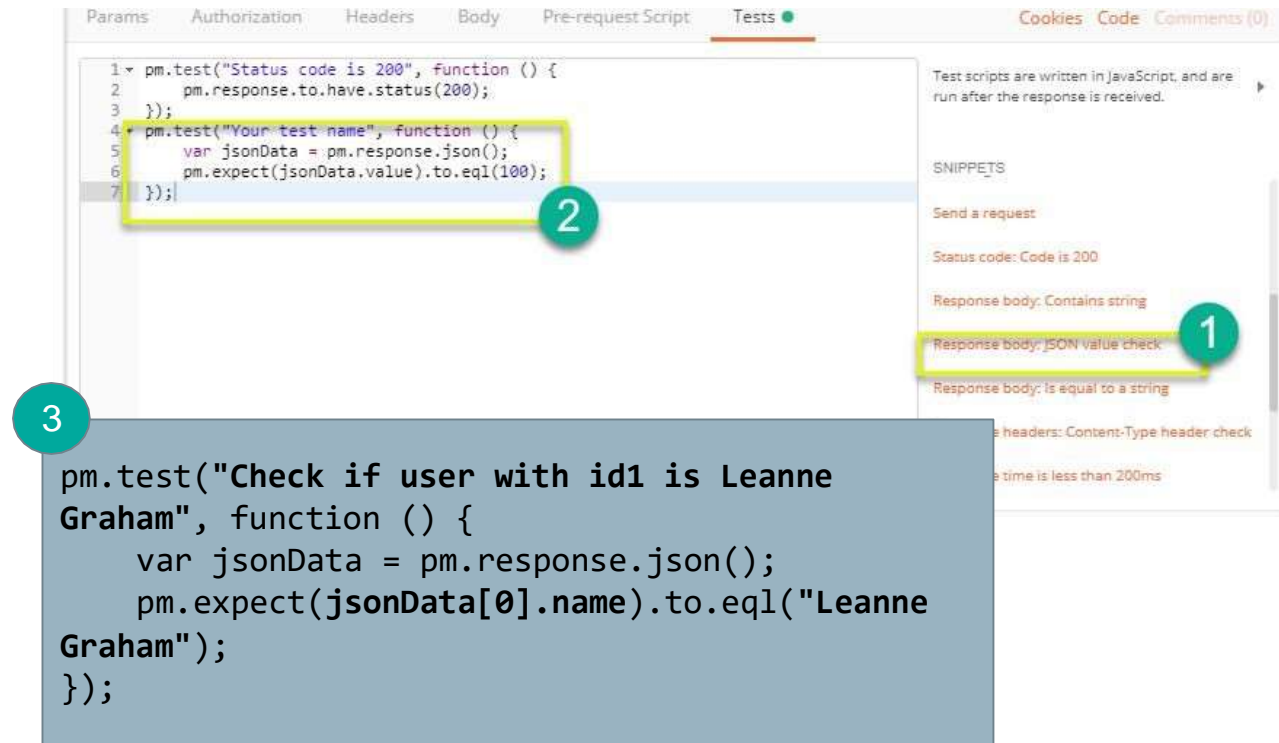
Test Execution Results

Both tests should pass. Status and test names indicated in GUI.

We Have Learned

- Unit testing focus on a single class.
- System tests focus on high-level functionality, integrating low-level components through a UI/API.
  - Identify an independently testable function.
  - Identify choices that influence function outcome.
  - Partition choices into representative values.
  - Form specifications by choosing a value for each choice.
  - Turn specifications into concrete test cases.

Next Time

- Choosing system test cases.
    - Handling infeasible combinations.
    - Selecting an interesting subset of specifications.

Thank You