CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 5: System Testing and Test Case Design

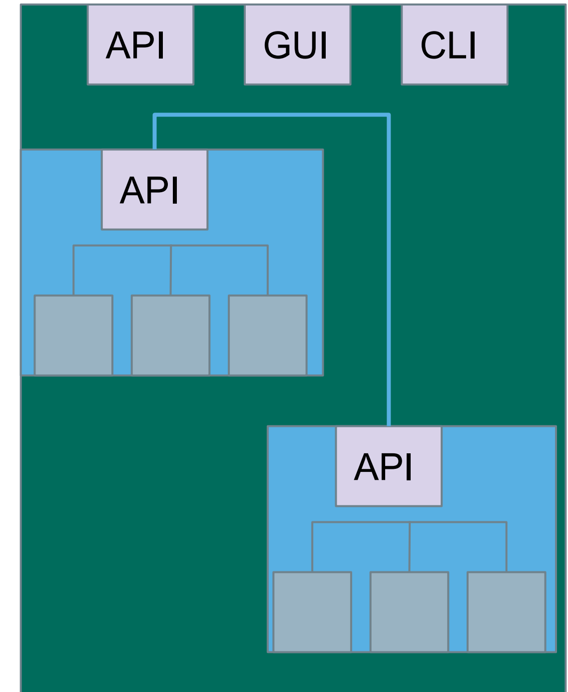
Gregory Gay
DIT636/DAT560 - January 29, 2024

Today's Goals

- Discuss testing at the system level.
- Introduce process for creating System Tests.
  - Identify Independently Testable Functions
  - For each:
    - Identify Choices
    - Identify Representative Values for Each Choice
    - Generate Test Case Specifications
    - Instantiate Concrete Test Cases

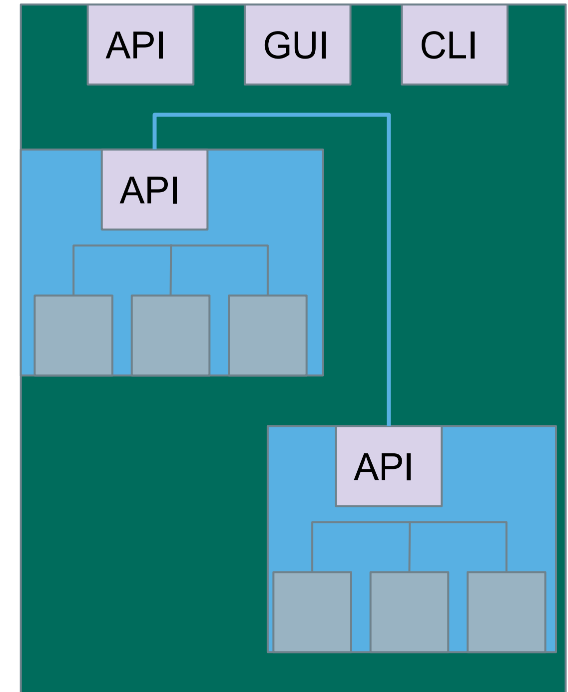
Testing Stages

- 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

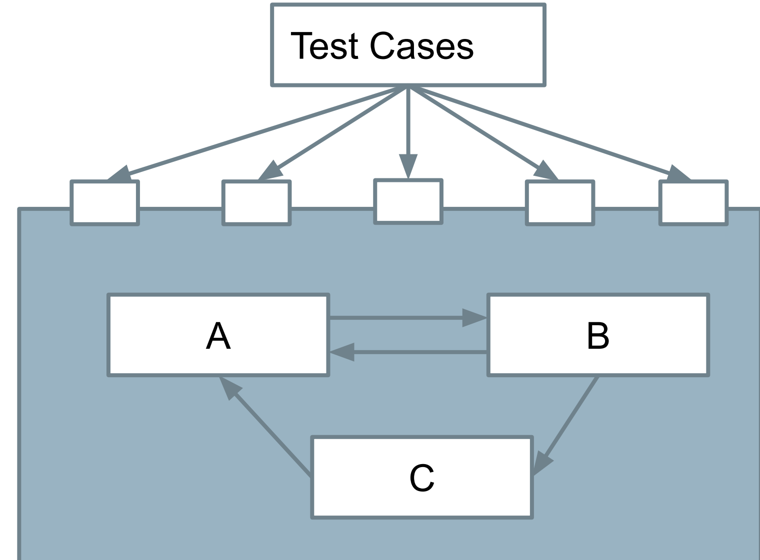
- **System-level Testing**
  - Tests **whole system** or **independent** subsystems through an **interface**.
  - **Integrates** lower-level components
    - (Subsystem-level) Do the collected units work?
    - (System-level) Does high-level interaction through APIs/UIs work?



System Testing

Subsystem made up classes of A, B, and C. Even if we have performed unit testing...

- Classes work together to perform subsystem functions.
- Tests applied to the interface of the subsystem they form.
- Errors in combined behavior not caught by unit testing.



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.
  - Can create tests using the requirements.
  - 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

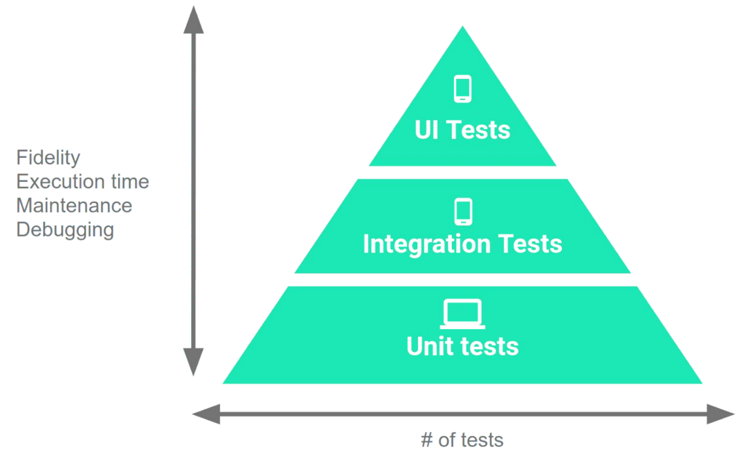
- Shared Memory Interfaces
  - A block of memory is shared between (sub)systems.
    - Data placed by one (sub)system and retrieved by another.
  - Common if system architected around data repository.
- Message-Passing Interfaces
  - One (sub)system requests a service by passing a message to another.
    - A return message indicates the results.
  - Common in parallel systems, client-server systems.

Interface Errors

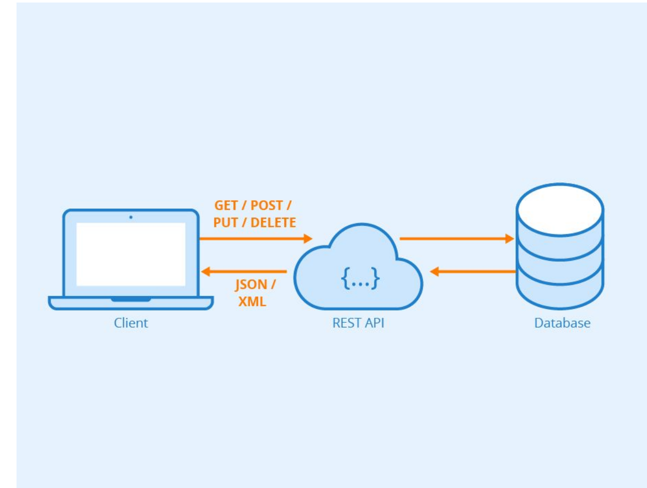
- Interface Misuse
  - Malformed data, order, number of parameters.
- Interface Misunderstanding
  - Incorrect assumptions made about called component.
  - A binary search called with an unordered array.
- Timing Errors
  - Producer of data and consumer of data access data in the wrong order.

Testing

- 70/20/10 recommended.
- Unit tests execute quickly, relatively simple.
- System tests more complex, require more setup, slower to execute.
  - UI tests very slow, may require humans.
- Well-tested units reduce likelihood of integration issues, making high levels of testing easier.



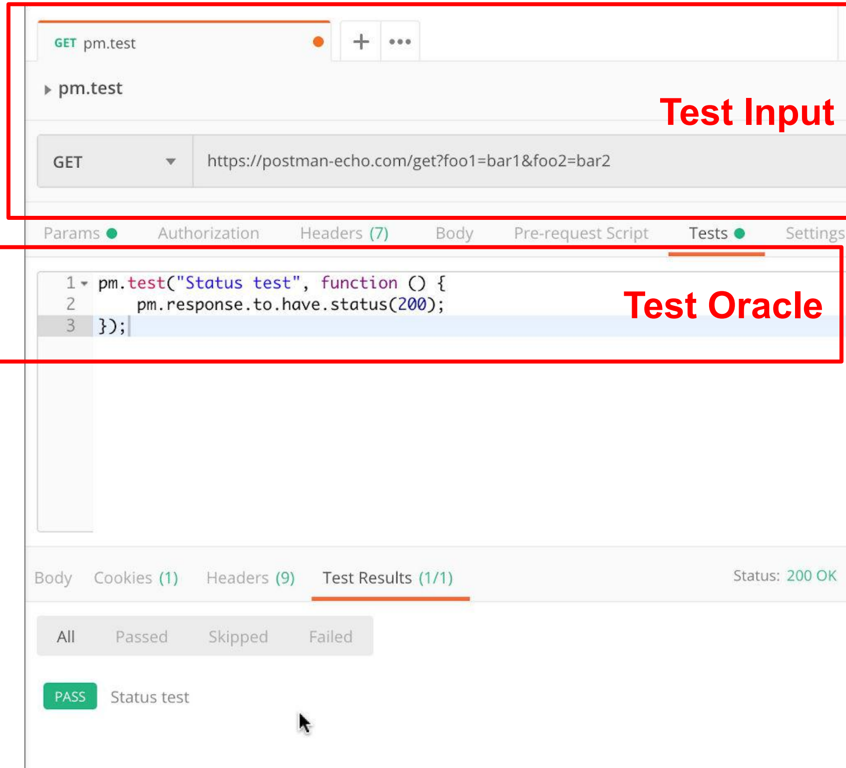
Creating System Tests for a REST API with Postman



Postman

- Testing framework for systems with a REST API.
  - REST: interface with **endpoints** we can interact with.
  - At an endpoint, we can send HTTPS request to:
    - **GET** information
    - **DELETE** information
    - **PUT** information into storage (ex: create a new entry)
    - **POST** information (ex: update an existing entry)
- Can create requests and tests using Postman.

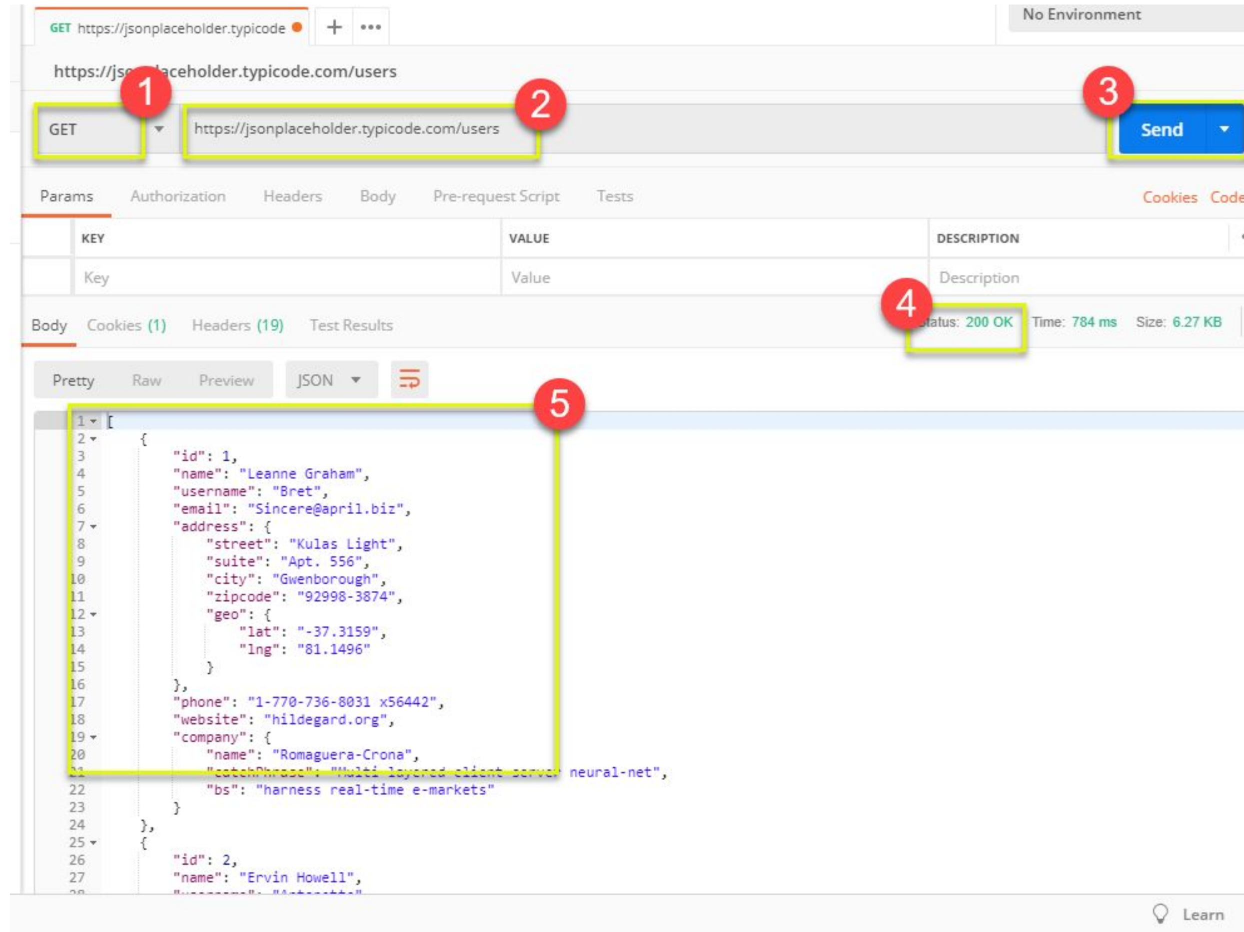
Writing Tests in Postman



- Each tab is a request.
- The request is the **test input**.
  - GET/POST/PUT/DELETE
  - Body, header, authorization, etc. for the request.
- Tests tab: **test oracles**.
  - Write small JavaScript methods to check correctness of output.

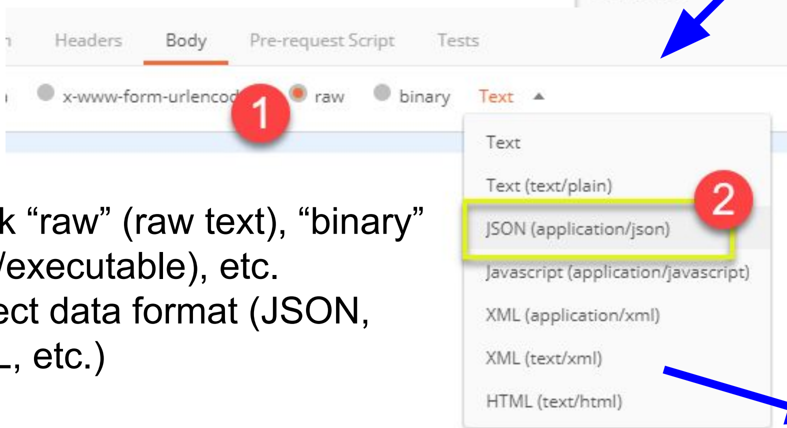
Input - GET

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.

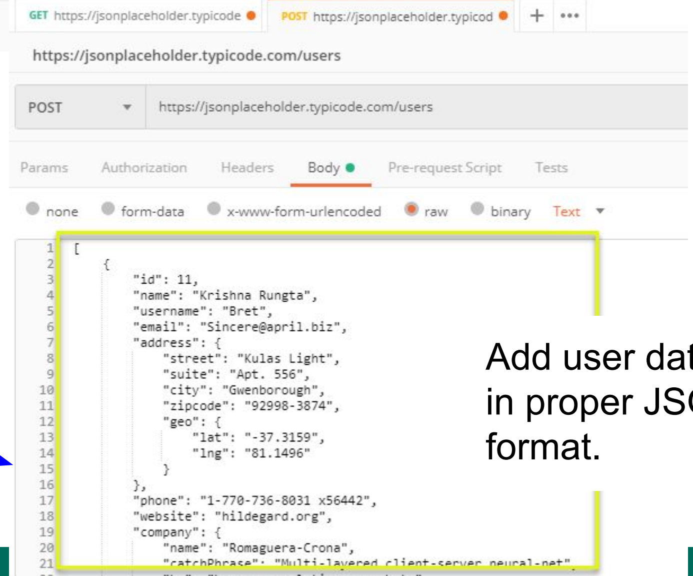


Input - POST

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.)

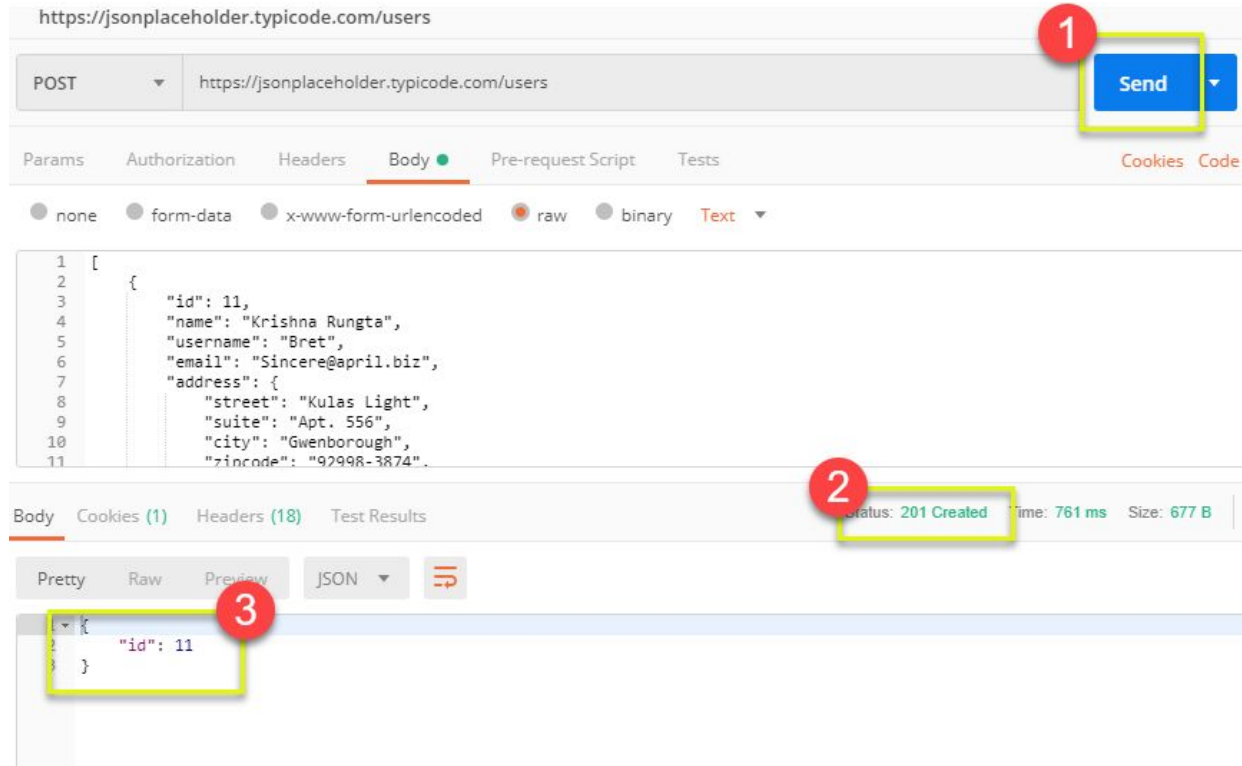



```
[
    {
        "id": 11,
        "name": "Krishna Rungta",
        "username": "Bret",
        "email": "Sincere@april.biz",
        "address": {
            "street": "Kulas Light",
            "suite": "Apt. 556",
            "city": "Gwenborough",
            "zipcode": "92998-3874",
            "geo": {
                "lat": "-37.3159",
                "lng": "81.1496"
            }
        },
        "phone": "1-770-736-8031 x56442",
        "website": "hildegard.org",
        "company": {
            "name": "Romaguera-Crona",
```

Add user data in proper JSON format.

Output - POST

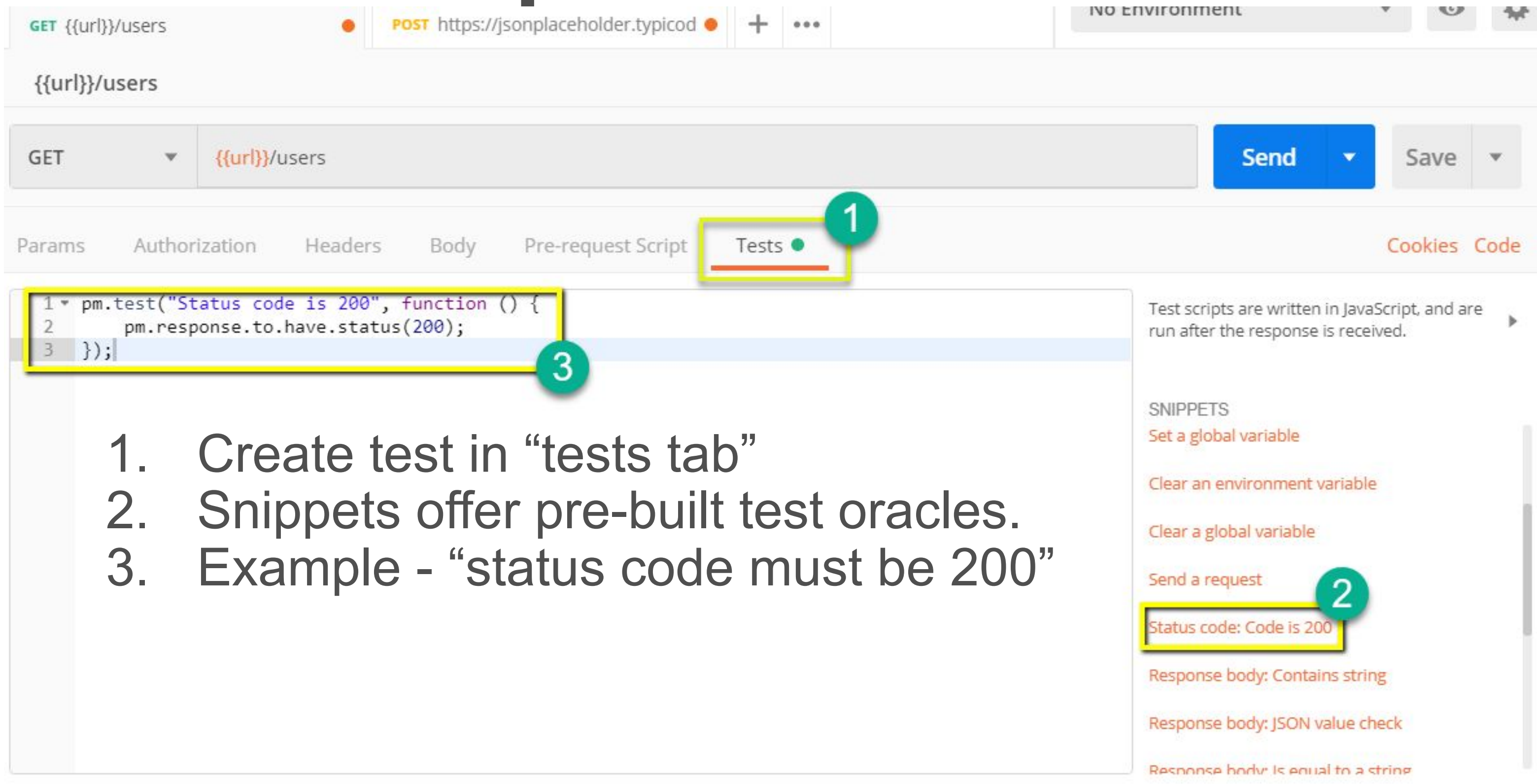
1. Click Send to send request.
2. Response status is indicated (201, data created)
3. Body indicates record “11” was created.



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.
- Use `pm.test` library to create assertions on output.
  - https://learning.postman.com/docs/writing-scripts/script-references/test-examples/ (many example scripts!)

Oracle Example - Status Check

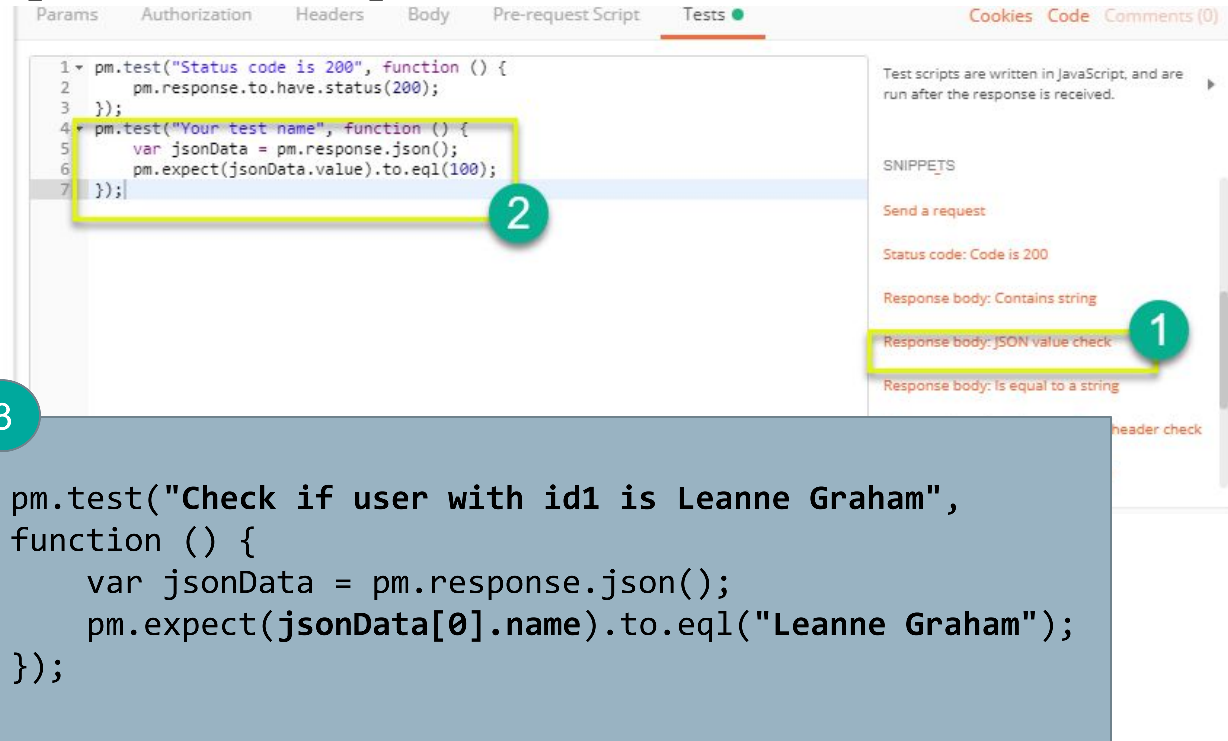


1. Create test in “tests tab”
2. Snippets offer pre-built test oracles.
3. Example - “status code must be 200”

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

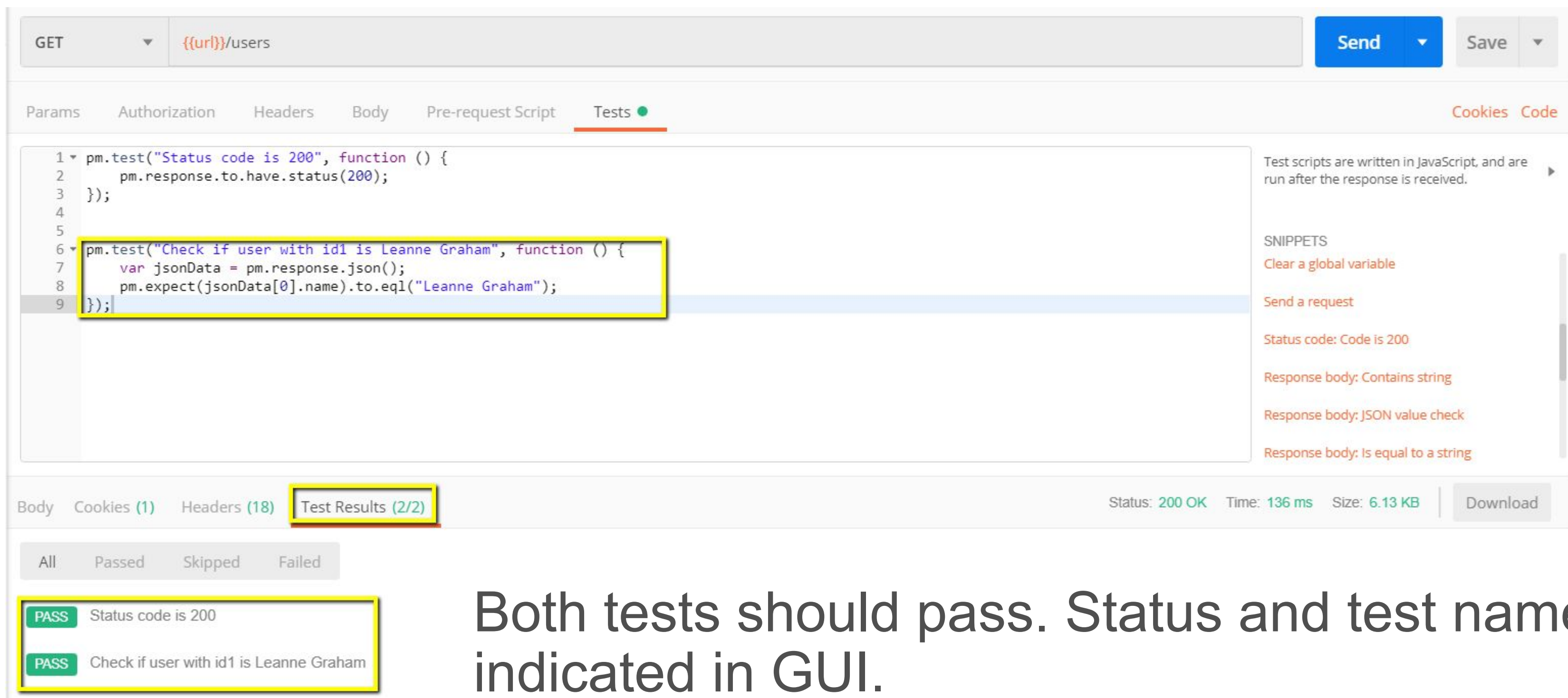
Oracle Example - Expected Value

1. Choose snippet "JSON value check"
2. This inserts generic test body.
3. 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");
});
```

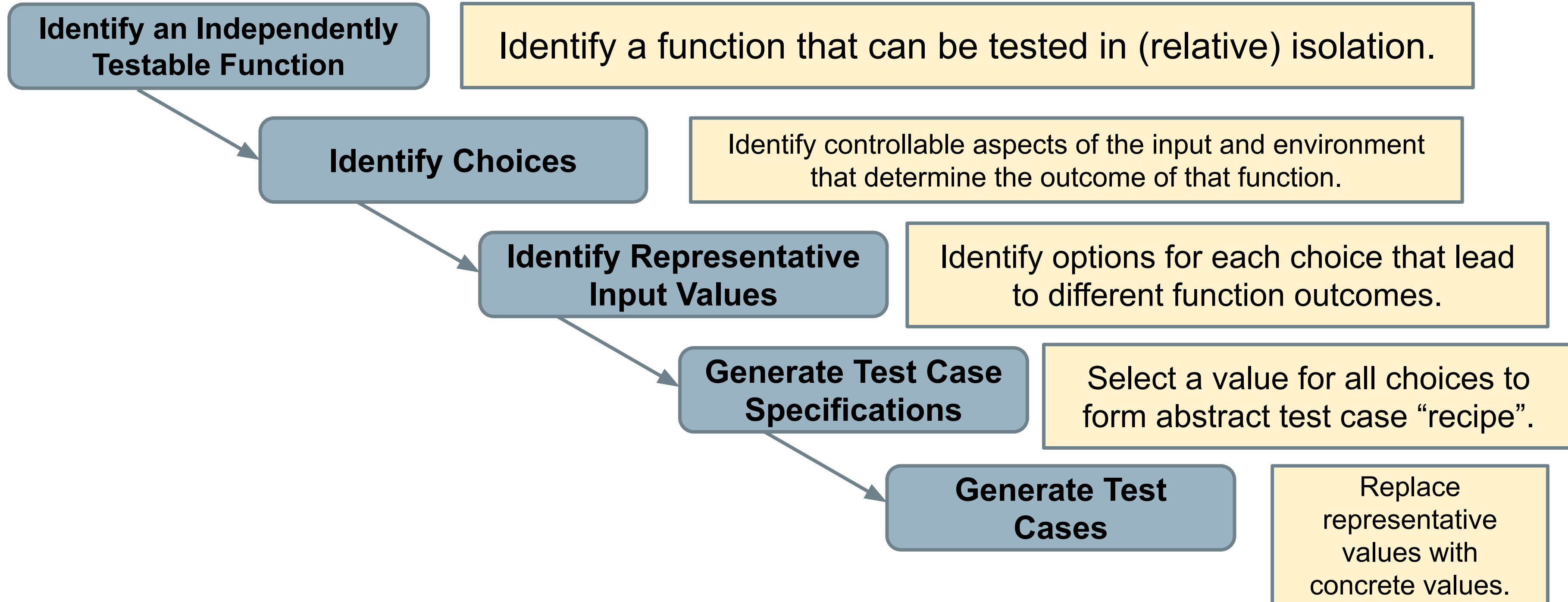
Test Execution Results



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

Creating System-Level Test Cases

Creating System-Level Tests



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)

Identify Choices

- What choices do we make when using a function?
  - **Anything we *control* that can change the outcome.**
  - What are the ***input parameters*** 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

- From the input parameters:
  - First Name, Last Name, Username, E-Mail Address, Password, Short Bio
- Other environmental factors:
  - Is there a database connection?
  - Is this user already in the database?

Register

Name *

First Last

Username *

E-mail *

Password *

Short Bio

Share a little information about yourself.

Submit

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.
  - A database affects User Registration, but there is **more than one** choice.
    - Choice: Is there a database connection?
    - Choice: Is there already a record for the user?
    - Choice: How full is the database storage?

Ex: Binary Search

```
Boolean binarySearch(String[] array, String toFind)
```

- **Choice: How many items are in the array?**
  - (Empty array might behave differently than one with several items)
  - (Could also provide a null pointer instead of a real array)
- **Choice: Is the array sorted?**
  - (Binary search assumes the array is sorted)

- **Choice: Is the string in the array?**
  - (Different function outcomes)

Let’s take a break.

Example

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

Input: Route: /registrations/, Method: POST,

Input: { “studentID”: VALUE, “courseID”: VALUE }

Output: Status Code: (201 if registration OK, 200 for input-based errors, others for other errors), JSON message: { “result”: VALUE } (“OK”, error messages)

Example Oracle:
```
pm.test(“Normal Case”, function() {
    pm.response.to.have.status(201);
    var jsonData = pm.response.json();
    pm.expect(jsonData.result).to.eql(“OK”);
});
```

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

```
Input: Route: /registrations/, Method: POST,
Input: { "studentID": VALUE, "courseID": VALUE }
```

- Does student meet prerequisites?
- Does the course exist?
- **What else influences the outcome?**

```
Example Oracle: pm.test("Normal Case", function() {
                    pm.response.to.have.status(201);
                    var jsonData = pm.response.json();
                    pm.expect(jsonData.result).to.eql("OK");
                });
```

Example - Register for a Class

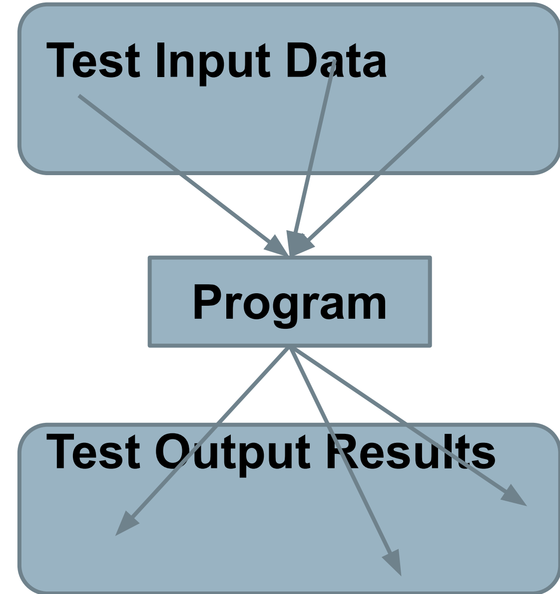
- During setup, we can influence a student's record and the course records.
  - These are "inputs" to consider.
- How are they used?
  - Has a student already taken the course?
  - Do they meet the prerequisites?
  - Does a course exist?
  - What are the prerequisites of a course.

Example - Register for a Class

- **Parameter: studentID**
  - **Choice:** Validity of Student ID
  - **Choice:** Courses Student Has Taken Previously
- **Parameter: courseID**
  - **Choice:** Validity of Course ID
  - **Choice:** Prerequisites of Course ID

Identifying Representative Values

- We know the functions.
- We have choices for each.
- **Representative values** are the options for each choice.



Ex: Binary Search

```
Boolean binarySearch(String[] array, String toFind)
```

- Choice: How many items are in the array?
- Choice: Is the array sorted?
  - Yes
  - No

- Choice: Is the string in the array?
  - Yes
  - No

- Choice: How many items are in the array?
  - Null pointer
  - 0
  - 1
  - 2
  - 3
  - 4
  - 5
  - …
  - 1000000000000

Ex: Register for Website

- "Value of X" are **choices**.
  - X = first name, username, etc.
- What are the **representative values** for each choice?
  - *First name could be any string!*

Register

Name *
First
Last

Username *

E-mail *

Password *

Short Bio

Share a little information about yourself.

Submit

Exhaustive Testing

Take the arithmetic function for the calculator:

```
add(int a, int b)
```

- How long would it take to exhaustively test this function?

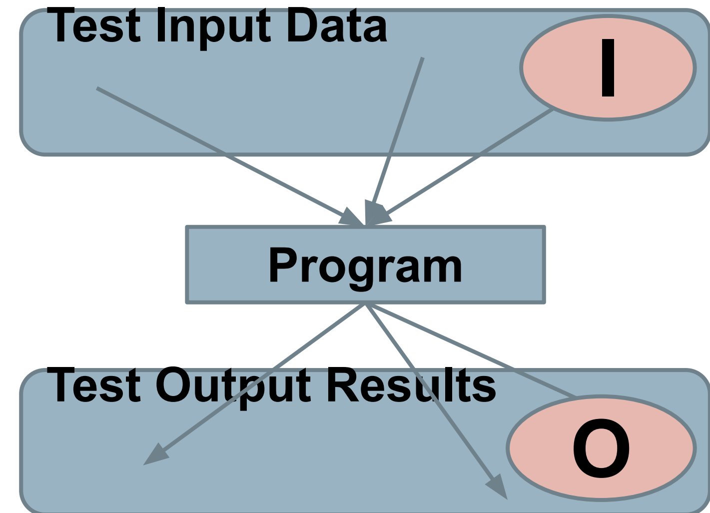
2^{32} possible integer values for each parameter.
= 2^{32} x 2^{32} = 2^{64} combinations = 10^{13} tests.

1 test per nanosecond
= 10^{5} tests per second
= 10^{10} seconds

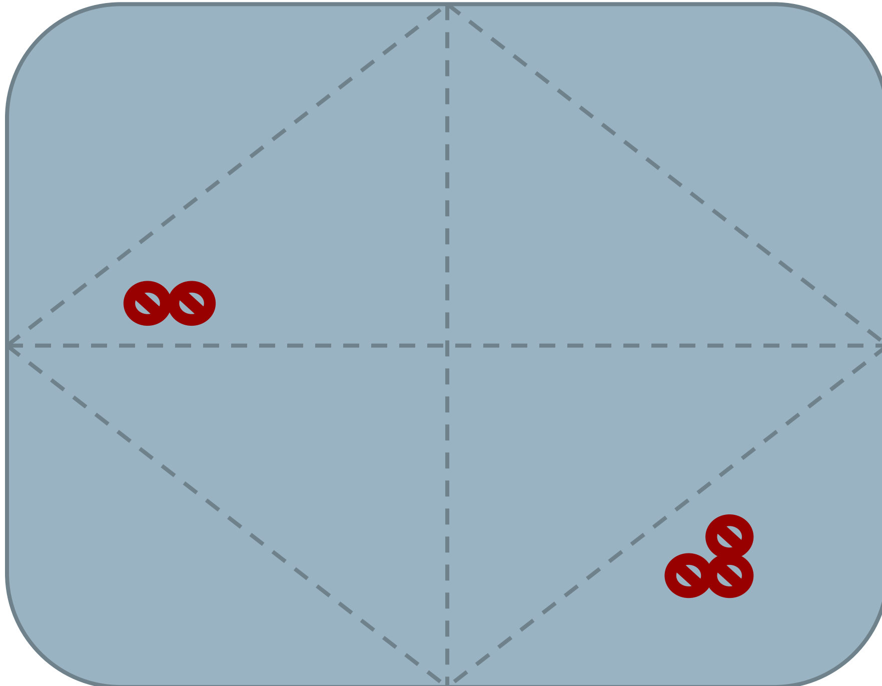
or… about 600 years!

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.



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.

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.

Equivalent Outcomes

- Look at the outcomes and group input by the outcomes they trigger.

```
Boolean binarySearch(String[] array, String toFind)
```

- **Choice: How many items are in the array?**
  - Null pointer
  - 0
  - 1
  - 2
  - 3
  - 4
  - 5
  - …
  - 1000000000000

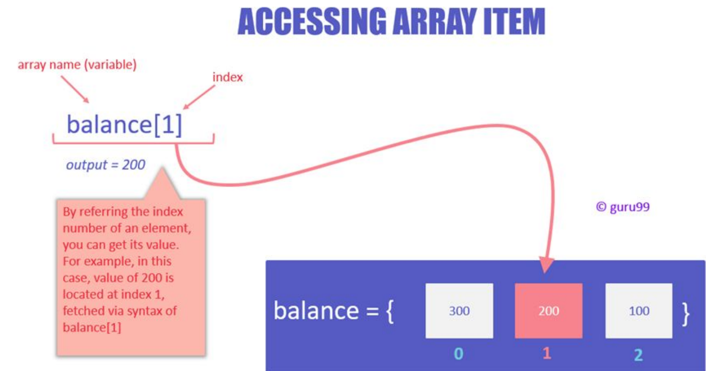
- **Choice: How many items are in the array?**
  - Null pointer (could lead to exception)
  - 0 (could lead to exception/warning)
  - 1+ (normal outcomes)

Data Type

- Try values commonly misused, based on data type.
  - Ex: Integer
    - Basic Split: < 0, 0, >0
    - If conversions take place from String -> Integer, use a non-numeric string.
- Also split based on how variable is used.
  - Integer intended to be 5-digit:
    - < 10000, 10000-99999, >= 100000

Data Type

- Data structures 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.



Data Type

```
Boolean binarySearch(String[] array, String toFind)
```

- **Choice: How many items are in the array?**
  - Null pointer (could lead to exception)
  - 0 (could lead to exception/warning)
  - **1 (single item collections often misused)**
  - **2+, # items == array size (normal outcomes)**
  - **2+, # items < array size (could be issues if array is not full)**

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)

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.

Quality Considerations

- Can add input partitions that help show that quality goals are met.
  - **Performance**: Input likely to lead to performance issues.
    - Ex: Remove resources, large input that will take awhile to process
  - **Security**: Input that attacker could apply.
    - Ex: Code injection in XML input.

Data Type

```
Boolean binarySearch(String[] array, String toFind)
```

- **Choice: How many items are in the array?**
  - Null pointer (could lead to exception)
  - 0 (could lead to exception/warning)
  - 1 (single item collections often misused)
  - 2+, # items == array size (normal outcomes)
  - 2+, # items < array size (could be issues if array is not full)
  - **10000 (could lead to performance issues)**

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`

Example - Register for a Class

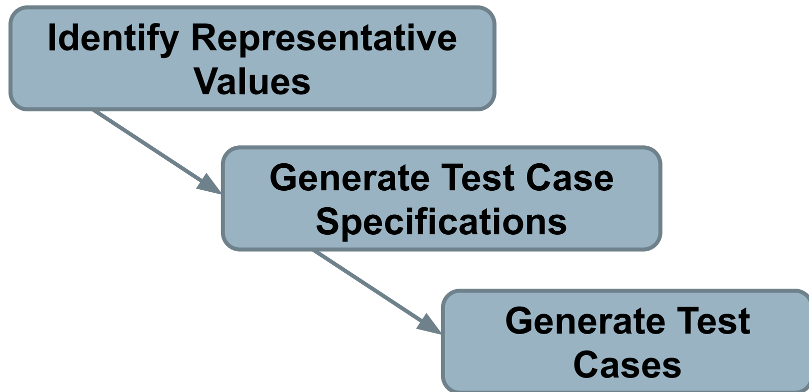
Parameter: studentID

- Validity of Student ID
  - Active Student
  - Inactive Student
  - Non-Existent Student
- Courses Student Has Taken Previously
  - Matches Prerequisites
  - Does Not Match Prerequisites

Parameter: courseID

- Validity of Course ID
  - Existing Course
  - Non-Existent Course
- Prerequisites of Course ID
  - Only Courses Taken By Student
  - Only Courses Not Taken By Student
  - Some Courses Taken by Student

Revisit the Roadmap



For each choice for a function, we want to:

1. Partition options for each choice into representative values.
2. Choose a value for each choice to form a test specification.
3. Assigning concrete values from each partition.

Basic Test Specification

```
// Set Up
    PUT /studentRecords/VALUE, {  … “status”: VALUE, “coursesTaken”: [VALUES]}
    PUT /courses/VALUE, { … “prerequisites”: [VALUES] }
// Attempt to register for a course
    POST /registrations/, { “studentID”: VALUE, “courseID”: VALUE }
// Check the result of registration
    pm.test(“Normal Case”, function() {
        pm.response.to.have.status(VALUE);
        var jsonData = pm.response.json();
        pm.expect(jsonData.result).to.eql(VALUE);
    });
```

Forming Specification

Parameter: studentID

- Validity of Student ID
  - Active Student
  - Inactive Student
  - Non-Existent Student
- Courses Student Has Taken Previously
  - Matches Prerequisites
  - Does Not Match Prerequisites

Parameter: courseID

- Validity of Course ID
  - Existing Course
  - Non-Existent Course
- Prerequisites of Course ID
  - Only Courses Taken By Student
  - Only Courses Not Taken By Student
  - Some Courses Taken by Student

Test Specifications:

- Active, Matches, Existing, Only Taken
- Active, Does Not Match, Existing, Only Not Taken
- Active, Does Not Match, Existing, Some Taken
- Active, - , Non-Existing, -
- Inactive, Matches, Existing, Only Taken
- Inactive, Does Not Match, Existing, Only Not Taken
- Inactive, Does Not Match, Existing Some Taken
- Inactive, - , Non-Existing, -
- Non-Existing, -, Existing, -
- Non-Existing, -, Non-Existing, -
- …

Specifications: 3 * 2 * 2 * 3 = 36 - Illegal Combinations

Generate Test Cases

Specification:

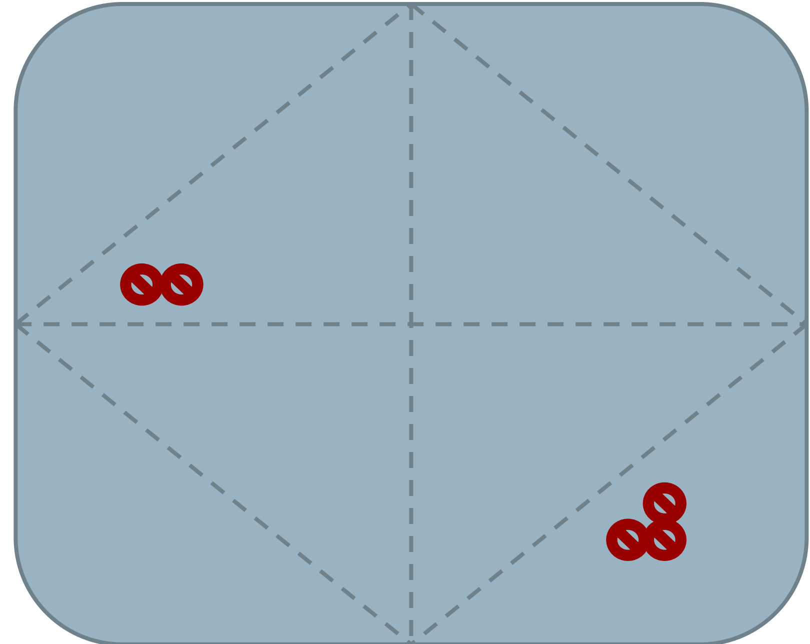
Active, Matches, Existing, Only Taken

```
// Set Up
    PUT /studentRecords/ggay, {“status”: active, “coursesTaken”: [“DIT050”, “DIT360”]}
    PUT /courses/DIT636, { … “prerequisites”: [“DIT360”] }
// Attempt to register for a course
    POST /registrations/, { “studentID”: ggay, “courseID”: DIT636}
// Check the result of registration
    pm.test(“Normal Case”, function() {
        pm.response.to.have.status(201);
        var jsonData = pm.response.json();
        pm.expect(jsonData.result).to.eql(“OK”);
    });
```

- Fill in concrete values that match the representative values classes.
- Can create MANY concrete tests for each specification.

Boundary Values

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

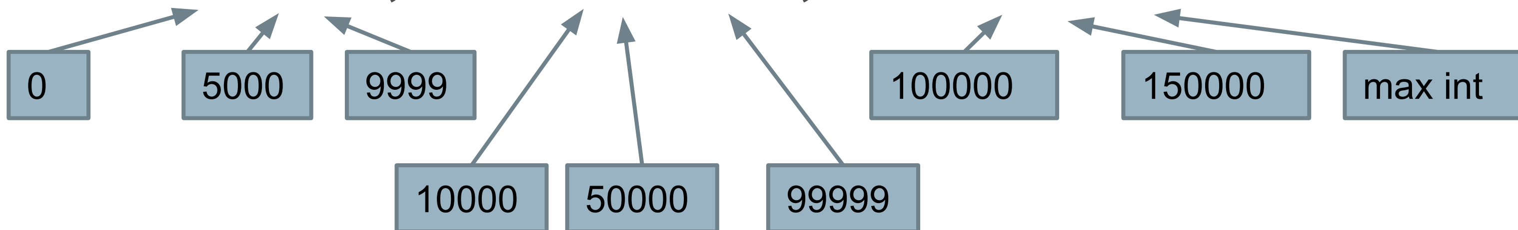


Boundary Values

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



We Have Learned

- 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

- Test Case Selection
  - Handling infeasible combinations.
  - Selecting an interesting subset of specifications.

- Assignment 1 - Due Feb 11
  - Based on Lectures 1-6

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY