GameBench API Documentation

Release 0.1.5

Big Fish Games, Inc.

May 06, 2019

Contents:

CHAPTER 1

Overview

The GameBench API Client library supplies a high-level object-oriented interface to the GameBench API. It is built in Python 3.7 and uses the Requests library and Pandas data frames to easily integrate into data analysis software.

The library has two main architectural components; the models and API packages. The API package is responsible for URL requests and dealing with the responses. The models are the objects representing the data returned. A mediator provides the glue between the api and the models.

As a user of the library, you should only ever need to interact with the models creator class and the model objects it can return.

Right now, the models are very thin. They only contain a property that has the data frame assigned. Over time we would like to add common functionality, like aggregates, to these classes.

CHAPTER 2

Using the GameBench API Client

2.1 Installation

The GameBench API Client can be install using pip:

```
pip install GameBenchAPI-PyClient-BigFish
```

It can also be found on the GitHub page.

2.2 Setup

Add the username and password for the GameBench account you are using to the 'global_settings.py' module.

```
GAMEBENCH_CONFIG = {
    'url': 'https://api.production.gamebench.net',
    'api_version': '/v1',
    'username': 'john.smith@example.com',
    'password': 'password',
    'company_id': '',
}
```

See *Global Settings* for for information on this module.

2.3 Basic Usage

In the module that will be calling the API Client, import the ModelCreator class.

```
from gamebench_api_client.models.creator.model_creator import ModelCreator
```

Create an instance of the ModelCreator and pass in the following information:

- The model that you want, which should be in a CamelCase style.
- A dictionary that specifies the information you want.

Here is one example for requesting time series data:

```
time_series_request = {
    'session_id': 66d926f47ff5a7a5d853d1058c6305614e1ae6a5
}

creator = ModelCreator('Cpu', time_series_request)
```

This will make the request to the API and store the returned data. You can call the *get_model* method for the *ModelCreator* instance to get the results.

See *Request Examples* for more dictionary examples.

CHAPTER 3

Request Examples

To make a request, import the *ModelCreator* class. Instantiating the *ModelCreator* requires two arguments. The first is a CamelCase style 'model' named after the metric that you are looking for; the model is dynamically imported based on this name. The second argument is a dictionary that must include specific key/value pairs for querying the GameBench API.

Here are examples of the dictionaries that can be passed to the *ModelCreator* as an argument to get back information.

3.1 Time-Series Model

This type of request is used for the following models: Battery, Cpu, CPU Core Frequency, Energy, FPS, FPS Stability, GPU (Imagination), GPU, Janks, Memory, Network, and Power. Here is an example of how you could get the CPU data for a session and what the DataFrame for that information looks like:

```
 from gamebench_api_client.models.creator.model_creator import ModelCreator


time_series_request = {
    'session_id': '66d926f47ff5a7a5d853d1058c6305614e1ae6a5'
}

creator = ModelCreator('Cpu', time_series_request)
cpu_time_series = creator.get_model()

results = cpu_time_series.data

print(results)

"""
      appUsage  daemonUsage    gbUsage  timestamp  totalCpuUsage
0  1372571.375            0  12.658228       5257      39.688461
"""
```

When requesting time-series data, pass in the model you want as the first argument to the ModelCreator. The given dictionary then just needs to include the 'session_id' key and the associated id as the value.

3.2 Generic models

This type of request is used for the following models: Keyword, Markers, Session Notes, and Session Summary.

```
generic_request = {
    'session_id': '66d926f47ff5a7a5d853d1058c6305614e1ae6a5'
}

creator = ModelCreator('SessionNotes', generic_request)
```

When requesting the session summary, markers, or notes for a session, the given dictionary only needs to include the session id. However, there are two other requests that can be made.

3.2.1 Keyword Search

This type of request allows you to pass in keywords that can be used to search for specific sessions. Adding a 'data' dictionary to the given dictionary will allow you to give keywords to search for.

Example:

```
generic_request = {
     'data': {
         'query': 'iPad'
     }
}

creator = ModelCreator('Keyword', generic_request)
```

3.2.2 Sessions

This type of request is different from the session summary request as it gives the summary information for multiple sessions. by passing in the appropriate key/value pairs, you can search for available sessions through the GameBench API. This type of request requires the 'session_id' key, that is used in normal session summary requests, to either not be included or have an empty string as a value.

Adding a 'params' key to the given dictionary will allow you to give search parameters.

Example:

```
generic_request = {
     'params': {
         'pageSize': 15
     }
}

creator = ModelCreator('SessionSummary', generic_request)
```

To see a full list of the available search options, see the GameBench API Documentation.

3.2.3 Session Detail

The session summary information also contains inner dictionaries, such as device information and app information. The *SessionSummary* class has class members which let you get just these details if needed. For example, if you just wanted information on the device that was used for testing just call the *SessionSummary.device* variable. This will return the device information in a DataFrame.

Here are all of the detail metrics you can call this way: app, device, location, metrics, and network app usage.

CHAPTER 4

How to Extend the Library

The GameBench API was originally written to use the Requests HTTP Library. However, this functionality can be extended to work with other libraries that need to be used. The following is an example of extending the Adapter for use with URLLib.

The abstract base class 'Adapter':

Listing 1: adapter.py

```
class Adapter(ABC):
    """ Abstract adapter for external HTTP request services."""

    def __init__(self):
        """ Abstract init that creates an empty service object."""

        self._http_library = None

    @abstractmethod
    def request(self):
        """ Abstract method to make a request."""

        pass
```

This class can simply be inherited to a new class for whichever library will be used. The _http_library instance variable should be used in the new class to hold a reference to the imported library. The request method needs to be implemented in all concrete classes and will be used to send the request. Here is the implemented class for using URLLib:

Listing 2: adapter.py

```
class URLLibAdapter(Adapter):

    def __init__(self, **request_attributes):
        super().__init__()
        import urllib.request  # Import the library needed
```

(continues on next page)

(continued from previous page)

```
        self._http_library = urllib.request  # Assign library to the _http_library variable
        self.method = request_attributes['method']
        self.url = request_attributes['url']
        self.headers = request_attributes['attributes']['headers']

    def request(self):
        return self._http_library.Request(self.url, headers=self.headers, method=self.method)
```

For this class we set up our local import of the urllib.request library. We assign that library to the instance variable to be used in the request method. The request_attributes argument will be the dictionary returned from the request retriever. This will include the needed elements for a request. Each of the needed needed elements is assigned to a variable. The request method then sends a request to the library using this information and returns the result.

Now that we have the new library set up to send requests, we just have to update our response retrievers to use this library. Here is what the Authentication Retriever currently looks like:

Listing 3: response_retriever.py

```
class AuthResponseRetriever(AbstractRetriever):
    """ Facade for getting Auth token from the Request.

        :param request_parameters: Dictionary containing information needed for
            an authentication request.  Example:
            {'username': 'John@gmail.com', 'password': '1234'}
    """

    def __init__(self, **request_parameters):
        super().__init__(**request_parameters)
        self.request = self.director.get_auth_request(**self.request_parameters)
        self.adapter = RequestsAdapter(**self.request)
        self.response = self.adapter.request()

    def get_response_json(self):
        """ Return the JSON of the response object.

            :return: The JSON data for a response.
        """

        return super().get_response_json()
```

This is using the RequestsAdapter to send requests. We can just update the adapter variable to use the URLLibAdapter to use the new library.

This is all that needs to be done to add in the library of your choosing. We'd love to refactor this class and allow it to be user configurable.

CHAPTER 5

Global Settings

The global_settings.py module is the home to the global variable GAMEBENCH_CONFIG. This dictionary holds the API endpoint information that the library uses. It also contains the username and password for the GameBench account you are using, and the company ID.

Listing 1: global_settings.py

```
GAMEBENCH_CONFIG = {
    'url': 'https://api.production.gamebench.net',
    'api_version': '/v1',
    'username': '',
    'password': '',
    'company_id': '',
}
```

If the API endpoint ever changes it can be updated in this file.

The company ID can be found on the GameBench Web Dashboard. Once signed in go to the 'My Account' section. The company id is the alpha-numeric string in the URL.

CHAPTER 6

Contributing

6.1 Who can Make Contributions?

We want everyone to feel free and welcome to add value to this project. We welcome all people from all roles and skill levels.

If you are a beginner, you might start with improving documentation, fixing lightweight bugs, little refactoring tasks, and improving tests. Please reach out to us and we will help you find approachable work.

Regardless of experience, we encourage you to talk through your plans with us and push branches early to get feedback before you get too deep.

6.2 What Code You Can Contribute

All code must be compatible with the license shipped with the project. This project is a BSD-3 licensed product. By contributing, you certify that you have a legal right to submit your contributions.

6.3 Contacting Us

We have a Gitter community setup for chatting through issues, feature requests, and all things software development.

6.4 The Development Environment

IDEs and text editors are very personal decisions. We want to you work with tools that you are effective using. We have supplied an EditorConfig file, configuration files for static analysis, and auto formatters, as well as our PyCharm .idea folder.

We love PyCharm, we recommend PyCharm, and we hope you use PyCharm and learn all its tools and shortcuts. If we can make one pitch, it is that we have PyCharm configured to use the same static analysis and formatting settings we check on push. Early feedback is always better than late feedback.

6.5 Project Management

We run a simple ZenHub Kanban board. You are welcome to work anything in the *backlog* at any time if it does not depend on incomplete stories. We prioritize the backlog every other Monday morning. Please find the highest priority item you would like to work on and commit to finishing within 2 weeks. While we are backlog grooming, we will look for items that have been in progress for 2 weeks and check on the status.

6.6 Version Control Process

6.6.1 Master

The master branch should always be releasable. Unreleased working software isn't supplying value to anyone. Our development standards and processes exist to help this be a reality.

6.6.2 Branching

When you begin work on a bug fix or feature, cut a branch from master. Be sure your local master is up to date with the remote master branch before creating yours. Name the branch with a clear concise name followed by the issue number. Ex: *URLlibSupport-#1123* or *FixDuplicateHTTPRequests-#213*

Update your branch from master often as you work to avoid complex conflict resolution at the end.

6.6.3 Commits

We recommend that you commit early and often. This gives you more freedom to go back and forth between commits as your solution evolves. However, the frequency of commits is a personal preference.

Commits messages should help you and others quickly understand what has changed and why. This can often be a way to track down new bugs or regressions. Finally, include a GitHub issue reference in all commits.

Summary: This should be a short title.

Example:

Interface Adapter for urllib.request.Request() #1234

Message: The message should call out what has changed and very briefly why.

Example:

Issue #1234

Added ULRLibRequest adapter class to match output from api.request.factory to input of urllib.request.Request() to keep interface segregation and decouple the library from our use cases.

Updated RequestTarget ABC to allow generic adaptees and specific adaptees in the adapter implementations.

Removed outdated inline comment from RequestTarget to improve readability.

6.6.4 Pushes

Push your branch to remote early in the process and then as you have working pieces complete.

The early push is to get your early design decisions out quickly. This allows you to share your branch with others for feedback even before a formal pull request and code review. Finding issues early is much easier to deal with than finding them later. This is an important consideration if you are new to our development approach or an inexperienced engineer.

Finally, you will create a pull request when you have finished. Pull requests have several automated checks that need to pass before we can merge your branch into master.

- The full test suite must pass.
- Docs generated.
- Static analysis must pass.
- Code coverage must be above 95%
- Compliance checks must pass.

In addition to the automated checks, an official maintainer must approve the pull request.

6.7 Code Review

Code review can be awkward for some engineers. It can be difficult to put your work up for judgment. Rest assured, future you will come to love it if you embrace it as an opportunity to learn and grow. Even when we disagree, we all come away with a broader perspective.

We all write imperfect code. If no one ever exposed us to different approaches, we would never grow. Submitter and reviewer must put aside their egos and help each other.

For reviewers, you are there to help the project and the submitter be successful. That means you must be clear and concise with your feedback. Candid, but respectful. You must also present a path forward. If you call out some code for improvement, we expect you to call out the standard it is violating and a little nudge in the right direction.

If either party violates our *Code of Conduct* , we will act as described in that policy.

Code review is a critical step in creating quality software. More eyes and minds are always better than fewer. Everyone who takes part can learn and grow, not just the submitter.

Code reviewers should focus entirely on whether the submission aligns with our development standards. Some elements might be subjective and not all code needs to perfectly align with standards. Everyone should strive for a consensus on what things the submitter must change, which should be new stories, and which are trivial.

There is a danger in kicking the can down the road when it comes to code quality. These things tend to snowball and drive down velocity over time. If a change adds certain value, it is worth doing now. Not later.

We will quickly reject pull requests that have any of the following anti-patterns and quality risks:

- No unit tests.
- No integration tests.
- Classes with low cohesion. If there is not a strong working relationship between the properties and methods.
- So-called “god” classes and methods.
- Many methods that exceed 10 logical statements.
- Many methods that need more than 5 arguments.

- Many methods that have many levels of indentation, such as nested if statements and nested loops.
- Copy/paste programming.

Free Open-Source Software (FOSS) is amazing because it brings together people who want to add value for everyone. We want you to succeed. So please push your branch early and ask for feedback if you see any of the items above appearing in your code.

A maintainer will merge the branch to master and release when you and the reviewers have reached consensus, fixed issues, and all automated checks have passed.

Next up, Development Standards!

CHAPTER 7

Development Standards

The value of software solutions is often clear and immediate. But many are surprised when their project's timeline and costs begin to spiral out of control. Sometimes you can slow this freefall so it only takes more use of the software to recover the costs, but it can also lead to disaster. We know this routine very well. And we are passionate about reducing these risks.

In the sphere of FOSS, it is important to live and breathe software quality. When we publish open source code, we expose ourselves to more unknowns. More clients, more use cases, and more reasons to change. For the project to remain relevant, it must adapt quickly and reliably while also adding value.

We also embrace the unknowns of a diverse group of contributors. Different skill levels, experience, and opinions on how to write software. When a project can align the power of diversity toward a common vision, awesome things can happen.

These standards are a guide to help the project meet its long-term goals and deliver value to the community. On this topic we have strong convictions. And we have experienced the costs of failure to follow them.

We also arn't perfect, so we write these down and focus code review on these standards.

These standards are also here so future you and other contributors have an easier time writing new features or fixing defects.

7.1 Code Style

We follow PEP-8 formatting with some modifications. The modifications bring formatting more in line with Google's Python style guide and some quality of life improvements. We also use Google style docstrings.

Please use our auto formatter configuration file. When in doubt, follow the conventions in the module you are working in.

7.2 Readability

Everyday writing is hard. Capturing the right tone, words, and structure to communicate clearly and concisely isn't trivial. And that's using a tool (written human language) that's been in active development for thousands of years.

Programming languages don't make communication any easier with all the symbols and *OddBall waysOf jamming_words_together*. How we name and structure our code has a profound influence on readability. And readability influences velocity and quality.

> "Indeed, the ratio of time spent reading vs. writing is well over 10:1. We are constantly reading old code as part of the effort to write new code. Because this ratio is so high, we want the reading of code to be easy, even if it makes the writing harder. Of course there's no way to write code without reading it, so making it easy to read actually makes it easier to write."

Martin, Robert C. The Robert C. Martin Clean Code Collection (Collection) (Robert C. Martin Series). Pearson Education. Kindle Edition.

In the context of FOSS, it is particularly important to consider the effect of poor readability on your community. Given an unknown audience of great diversity, we ought to make special effort to ensure the meaning of our code is self-evident.

Here are some rules of thumb we like to follow:

> **The Sentence Check:** Read your code out loud. If someone not familiar with the project could understand you, it should be good. This is easier with higher level policies and interfaces than with low level details, of course.
>
> **Universal Readability > Idioms > Conventions:** Prioritize universal understanding over idiomatic code or conventions.
>
> Many languages have unique ways of doing things, or idioms. If those idioms interfere with readability and supply no other significant value, avoid using them. A one-line Python list comprehension can be beautiful. But, can quickly turn into a tangled mess if it grows beyond a single line.
>
> Many conventions exist to serve needs that are outdated or unnecessary. We will touch on common conventions later.
>
> **Comments are Red Flags:** Specifically, comments that describe *what* the code is doing show the code isn't expressive enough. If the code is expressive, then the comment will be redundant and become another axis of change, defects, and waste.
>
> This guideline doesn't apply to docstring comments or comments that describe *why* a contributor took a specific approach. When we write controversial code, a good comment describing why we wrote it is especially useful.

7.2.1 Names

As a software engineer, you spend much of your time reading code. When you aren't reading, you will be naming things. Variables, methods, classes, modules, packages, and tests. The quality of your names has a direct effect on the productivity of your peers and future you.

A name should communicate what, how, and why in as few words as possible.

7.2.2 Variables

Variable names should be nouns separated by underscores. Either singular or plural depending on context. You can usually name a list of employee objects, *employees*. Context, such as scope and the variables within the scope,

can inform other decision about variable names. Variables should be more descriptive and verbose when they occupy a broader scope. Global variables should then supply all the needed context within their name. Consider: *globals.DATABASE_TIMEOUT_IN_SECONDS* vs. *database.connection.settings.timeout*. Both timeout variables are appropriately descriptive within their scope.

You should avoid encodings, prefixes, suffixes, and pointless conventions. This includes:

- Type "hints" in the name. If you have two collections in the same scope that hold the same things but in different collection types, then please use some typing information in the name. Otherwise, the type "hint" becomes noise and in some cases disinformation. The variable *full_name_string* is no more informative than *full_name*. Neither is *employee_object* more informative than *employee*.
- Conventions like Hungarian notation and *m_* for members do not add value.

Conventions that can help:

Single letter variables can work well in little algorithms that carry history and meaning. For example, incrementors can be, "*i*." And formulas can use their common variable letters, such as:

In Python:

```
a = 3
b = 5
c = math.sqrt(a \*\* 2 + b \*\* 2)
print(str(c))
```

However, you may also consider using more descriptive names for those without specific knowledge.

Example:

```
adjacent = 3
opposite = 4
hypotenuse = math.sqrt(adjacent \*\* 2 + opposite \*\* 2)
sine = opposite / hypotenuse
```

You may also use the common access level intention hints that are common in the Python community. A single leading underscore to communicate that clients shouldn't use this member outside of the class. If it is important that clients do not easily access the member, go ahead and name mangle it with a leading double underscore.

Please use all caps for constants.

7.2.3 Classes, Modules, Packages

All high-level collections should also be nouns. The higher-level you go, the more bias you should express toward using problem domain names. The lower level you go, bias toward solution domain names. Looking at the root of your project, you should be able to clearly understand what problem the software is trying to solve. As you click into the folders and then the modules you should see more of how the engineers have chosen to solve the problem.

Example:

- gamebench_client
  - api
    - api_facade.py
    - entity_request_mediator.py
    - requests
      - interfaces

            - requests_library_adapter.py
            - request_builder.py
            - parameters
            - parameter_builder.py
            - parameter_director.py
            - methods
            - method_factory.py
            - urls
            - url_builder.py
            - url_director.py
        * entities
            - entity_factory.py
            - metrics
            - metric_factory.py
            - cpu.py
            - gpu.py
            - sessions
            - session.py
    - settings
        * defaults.py
        * custom.py

When using a design pattern or common structural elements, use the common names for these. Examples: *URLLibAdapater*, *RequestBuilder*, *UserModel*, or *UserView*.

Packages and Modules

Use all lowercase words separated by underscores. Singular or plural based on context.

Classes

Use *CamelCase* nouns. Singular or plural based on context. The more specific a class becomes, the more verbose its name will become. Consider *User*, *EmployeeUser*, *AcmeCorpEmployeeUser*. Each covers a specific set of behaviors and data and the name becomes more verbose. If you use a class far away from its definition or the path does not supply good context, you should also consider a more descriptive name over a concise name if refactoring the structure isn't practical.

Methods

Methods are actions, so they should be verbs. Use lowercase underscore separated words. A method should either ask a question (query), ask for data (query), or issue a command that changes state. A method should not be both a query and command. So be sure that your name matches the action type to avoid confusion.

Query and Command Separation:

```
class User:
    def __init__(self, first_name, last_name):
        self.first_name = str.strip(first_name)
        self.last_name = str.strip(last_name)
        self._is_active = True

def get_full_name(self):
""" Query the full name of the user."""
    space = " "
    full_name = space.join([
        str.title(self.first_name),
        str.title(self.last_name)
        ])

    return full_name

def activate(self):
"""Change state of user to active."""
    self._is_active = True

def deactivate(self):
"""Change state of user to inactive."""
    self._is_active = False

def is_active(self):
"""Query user's active status."""
    return self._is_active
```

Avoid This:

```
def activate (self):
"""Change state of user to active and return user._is_active."""
    self._is_active = True

    return self._is_active
```

In command methods, use exceptions to communicate failure rather than different returns.

Be consistent when using words that have little distinction. For example, what is the difference between get and fetch? We default to using "get" for query methods until we find another name with a meaningful and unambiguous distinction that fits the context. One might argue that get_full_name() isn't a pure getter. We tend to agree. Someone might name it, concatenate_first_and_last_names(). The problem is that "concatenate" sounds like a command that changes state and does not return anything.

There is processing but no state change and we want the full name returned. We are preferring *clarity of intent* over perfect accuracy or conformity to convention. Depending on the scope of this methods use, you might even name it my_user.get_first_and_last_as_full_name().

The Python standard library doesn't always do this. For example, *str.join()* returns the joined elements of the iterable given in the arguments into a string using the string object calling it as the separator. It is a confusing method if you aren't familiar with it. It sounds like a command but returns data and the caller occupies an odd role. For greater clarity, a *to_string()* method on the iterable class and/or *from_iterable()* on the string class. Ideally, both would be available.

To get a sense of clarity, try reading the following code out loud:

```
full_name = space.join([
    str.title(self.first_name),
    str.title(self.last_name)
    ])
```

"Full name equals space join stir title first name and stir title last name." The choice to use "str" is historical and a barrier to entry for those new to our field. As is "char," "int," and all other abbreviated words. Consider these alternatives:

```
def get_full_name(self):
    full_name = self.first_name.get_title_cased() +
        " " +
        self.last_name.get_title_cased()

    return full_name

def get_full_name(self):
    full_name_items = [
        self.first_name.as_title_cased(),
        self.last_name.as_title_cased()
        ]
    full_name = String.create_from_list(name_elements, separated_by=" ")

    return full_name
```

The first reads: *"Full name equals first name get title cased plus space plus last name get title cased."*

The second reads: *"Full name items equals first name as title cased and last name as title cased. Full name equals a string created from list of full name items separated by a space."*

Both examples are much closer to spoken English. And you can see there is quite a bit of latitude to carefully consider names. With just a few *minor* interpretations, these really could read as a normal sentence. *"The full name is a string created from a list of name elements separated by a space."* As you write your code, try to find ways to reduce the amount of interpretation needed and your intent will become clearer.

Examples:

my_string = some_list_object.to_string([separator]) which would return a new string object.

my_string = String.from_iterable(iterable[, separator]) This would be an override of *String.__init__()*.

This isn't a judgement of Python. We love Python and enjoy it the most of all interpreted languages. We also recognize that this would be a substantial change. We also recognize the historical context that this appears to come from.

The method, *str.join()* is a good example of how names and structure can influence the learning curve through readability. When learning curves are low, everyone can be more effective more quickly.

You will also notice in the query and command separation examples there are two setters for the active property. We aren't against setters as a convention, but if there is an opportunity to make code more semantic, we like it. In this case, calling a method with no arguments and a name expressing clear intent is a straightforward way to reduce silly bugs and increase readability.

```
new_hire = Employee.from_full_name(full_name)
new_hire.set_active(True)
new_hire.save()
```

vs:

```
new_hire = Employee.from_full_name(full_name)
new_hire.activate()
new_hire.save()
```

even better, but this comes later.

```
human_resources_facade.onboard_new_hire(full_name)
```

7.3 Automated Tests

We passionately believe quality automated tests are critical to the success of any software project.

> "The third of W. Edwards Deming's fourteen points for management states, 'Cease dependence on inspection to achieve quality. Eliminate the need for inspection on a mass basis by building quality into the product in the first place' (Deming 2000). In continuous delivery, we invest in building a culture supported by tools and people where we can detect any issues quickly, so that they can be fixed straight away when they are cheap to detect and resolve."

Forsgren PhD, Nicole. Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution Press. Kindle Edition.

We also believe that how we distribute our effort across the test levels can have profound effects on the usefulness and value of the tests. We solve for how to spend effort by understanding when testing creates value. Finally, we need to understand what parts of the process of development, testing, and fixing defects are the most expensive.

Google has published a nice little article on this topic: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html

7.3.1 Ideal Distribution of Tests

The lower level the test, the faster it runs, the more quickly you get feedback, and the defect is more isolated. This means faster fixes. Aim to have 70% of your tests be unit level tests. A unit is simply the smallest container of organization. In our case, that is a method or function.

Pull requests must have all *applicable* test levels covered before maintainers merge it to master.

7.3.2 Coverage

The ideal coverage is 100%. With a language like Python, 100% can be possible. CI checks enforce 95% coverage from the entire suite. You do not need to add unit tests for these things:

- Objects that you cannot call directly.
- Trivial methods. Methods that have no branching logic and only use the simplest of statements. Example:

```
def get_name(self):
    return self.name
```

- 3rd-party libraries, unless you suspect they are defective.

7.3.3 Recommended Approach

There are competing approaches to writing tests. Given the goal of fast feedback and the Lean principle of building quality into production processes, test-driven development is the clear winner. We have noted some confusion about what is and isn't test-driven development (TDD).

Here is the short story:

- There is no single magic potion to make clean higher quality code. You must exercise multiple disciplines to reach the goal. TDD is just one of them.
- TDD as we recommend practicing it, is super Lean. *TDD has several cycles* that cover the entire scope of the application from line-to-line all the way up to architectural considerations.
- TDD doesn't mean you can skip planning and design altogether. Have a big picture in mind. But don't force your design to work. Through the full TDD cycle, you will gain information that will better inform your structure.

7.3.4 Test-Driven Development (TDD):

When we talk about TDD, we are talking about a multi-cycle process. The cycles of TDD have distinct goals. An engineer must use all the cycles well to realize the full benefits of TDD.

Second-by-Second:

The focus here is brutally simply: *Make it work!* And at *any* cost. You will move through this cycle several times before a unit test is complete.

1. Write a failing test before any production code.
2. No more test code than you need to create a failure.
3. No more production code than you need to pass the failing test.

The tests you write at the beginning of the second-by-second cycle will be ugly. And the production code you write will be too. Just make them pass. Don't resist this.

If you get stuck at any point in this cycle, either because you cannot make the test pass or because you feel like you must write a bunch of untested production code, your test is wrong. The test definition might be incorrect, or the test covers too much of the problem. *Think smaller and focus on what you know first.*

Minute-by-Minute:

This is the first step where you will begin to focus on making it "right." You will move through the complete cycle about once per unit test.

The refactor step is a common point of failure. Sometimes we speed right past it and then only later realize our code is gross. Always keep your eye out to remove duplication and improve readability once your code works, but not before. *Focus* on one problem at a time. Moving on to make a new unit test pass without refactoring only makes the work to clean it up bigger and riskier. And that's not Lean.

Refactoring must be a continuous activity part of the minute-by-minute process of creating software if we are to make *software that works and can change easily*. There is not going to be time later. There is no tomorrow. There is only now…

10 Minute Cycle – Generalize:

Here we apply even more effort toward making it right. Within this cycle, you will be applying the Transformation Priority Premise. Look for overly specific production code and then generalize it. Your tests and test suite should become more specific and detailed, but your production code should also become more general.

Here, you might again find yourself stuck. If that happens, you need to start removing tests until you can take a different approach with different tests.

Hour-by-Hour – Architecture and Design:

Finally, we make things right within the context of bigger design buckets. A focus on the small or local problem can blind you to the fact you are crossing architectural boundaries or violating SOLID principles. You should take time every hour or so to analyze your design, but don't make changes unless all your tests are passing.

We will talk about architecture and object-oriented design in a later section.

7.3.5 Scope of a Unit Test

In traditional QA language, a unit is the smallest measure. Typically, this would be a method. Following TDD, you will make your production code increasingly general as your test become more specific. This means you start out with a unit test covering one method but then you break that method into smaller units as you extract out logic. That original unit test is still covering the refactored code. This is a decision point.

One path would suggest that you mock out the behavior of the new methods for isolation. Another would be to ensure that your current unit test is supplying all the behavioral coverage needed and stop there. The correct response is a judgement call. And there are trade offs. The more you mock, the more you couple your tests to implementation. The less you mock the less isolation you achieve.

There are real dangers to both. Coupling tests to implementation can lead to overly fragile tests. But insufficient isolation can reduce the benefit of unit level testing. The ideal test pyramid is a guide to help you, so are your fellow engineers. So, talk it through, weigh the risks and benefits, and decide.

7.4 Documentation

If we write our code well, how we intend others to use it should be as self-evident as how it works. That's a nice goal, but let's not lean into that too much. Many of the words we use in programming are overloaded or not universally understood. Our documentation aims to reduce those risks while reducing the risk of spending too much time on documentation that might change too often to be useful. Here is our approach:

- We write docstrings as part of the development process for everything but the most trivial code. We include everything a client would need to know to use the code and what to expect from it.
- We write or update user documentation as needed. We include only what a user needs to understand what we believe is the ideal way for them to use the software. For example, a user should only ever need to interact with the entity factory method and the objects it returns to them. So, we document this method and the objects it returns with realistic use cases and examples.
- We also document where we have designed the software to be extendable. For example, if you can only use a specific HTTP library for security reasons, we will supply examples of how you can replace the Requests library.
- We document the parts of the software that you can change through configuration settings.

As you work on features and bug fixes, you will be creating and updating docstrings. You should also cross-reference the documentation to be sure that your changes don't create buggy documents.

All documentation must be in a format understood by Sphinx. Please use reStructuredText as the markup language. See the "Resources" section for links.

7.5 Object Oriented Design and Software Architecture

Our goal with design and architectural decisions is to reduce coupling and improve understanding of the system. To create working software is not enough. We must also create software that is easy to change and adapt to different environments.

7.5.1 Object Design

How we design our objects influences how easily others understand our software and how easy it is to change. As an engineer gains more skill, they will learn diverse ways to exploit object-oriented techniques to achieve these goals.

The Starting Point: Cohesion

To begin, our classes tend to reflect reality in some logical way. A car class has properties like make, model, year, and color. And methods like accelerate and brake. As a start, this is okay but quickly starts to fall apart as we try to model this class closer to real cars. For example, not all cars have four wheels, 5-speed manual transmissions, or run on gasoline. Most cars steer with just the front two wheels, but some have 4-wheel active or passive steering.

As we add more configurations, these configurations then introduce changes to what behaviors are possible and how you implement those behaviors. If we stay in this mode of thinking, we will up end with a giant class full of if statements, huge methods, and bugs.

If you are new to OOP, we suggest thinking about class design in terms of cohesiveness rather than a logical reflection of the real world. This should lead to more but smaller highly focused classes and fewer parameters in constructors and other methods.

Look for these warning signs:

- Methods with more than 3 or 4 parameters. These might reveal to you a class in hiding.
- Methods that don't act on any properties or supply a high-level interface to other methods in the class. These might reveal to you feature envy. The method might better belong in another existing class or a new class.
- A group of methods act on a group of properties and another group of methods on another group of properties. These might reveal to you a class in hiding.

The Next Step – SOLID

It is helpful to have a framework of principles to guide us toward highly cohesive classes and make design decisions that help us reduce rigidity and fragility in our software. We like Dr. Martin's collection of object-oriented design principles for their relative simplicity and the easy to remember acronym.

Single Responsibility Principle (SRP)

A class should have one and only one reason to change. An easy illustration is a report. The format of the report and the calculations that create the data within it are distinct reasons to change. These responsibilities should live in different classes.

Open-Closed Principle (OCP)

You should be able to extend a class' behavior without changing the class. For example, you should be able to add support for Python's urllib without changing any code related our implementation of the Requests library. Gamebench-Client.api.requests is open for extension. You would implement a new concrete adapter inheriting from the adapter's ABC and enter a custom setting for request library path.

The result of designs that conform to this principle is the ability to add new features without changing already working code.

Liskov Substitution Principle (LSP)

Methods that use or refer to a base class must be able to use objects of derived classes without knowing anything has changed. This one can be obvious when you have a bunch of if, elif, else statements figuring out how to work with derivatives. However, you can violate this one more subtly too. If you find yourself changing the base class definition to accommodate a new derivative, it is a sign you might be violating this principle.

Interface Segregation Principle (ISP)

You should not force a client to depend on methods it does not use. What this means is that you should be defining small interfaces specific to a client. If you find that your interface has methods used by one client and a group of methods used by another, split them out into different interfaces. For example, we don't want our interfaces that handle creating URLs and other request parameters to know anything about the Requests library's interface. We certainly don't want an import dependency. So, we have used the adapter pattern to conform the two interfaces. If we need to conform another HTTP library to our module, we simply derive another adapter.

This principle further guides us toward pluggability and software that is simple to change.

Dependency Inversion Principle (DIP)

Always depend on abstractions, not concrete classes. If you consistently apply OCP and LSP, you will arrive at this dependency inversion principle. You will know you are violating this principle when you change one thing and must change many others, you make one change that breaks other areas that might even be unrelated, and you cannot reuse your module in another piece of software.

One of the places we see this principle most violated is in web applications that use an ORM. If your core business logic has direct dependencies on your framework, you couple the code that has the most value to a trivial detail. The cost to change is incredibly high. The solution is to provide an interface that allows you to swap ORMs with relative ease and keep the ORM imports out of your critical business logic.

We measure abstractness as the distance from I/O. There are other ways to think about it, but this is the most straightforward way we have found. The closer a module or class is to inputs or outputs, the more concrete it is. A file reader/writer, HTTP request/response, and device drivers are good examples of the most concrete details.

This principle applies between classes, modules, and the entire architecture of the software.

Step Three: Common Problems w/ Common Solutions

After some time designing objects, you will find common problems that you will need to solve. A frequent problem ought to have a common solution. This is where design patterns come in to save the day. Diving into design patterns is outside the scope of this document, but you will see solution domain names related to design patterns all throughout this project. You will also likely run into a reviewer asking you to use a certain pattern when creating a new feature.

We encourage you to study all the design patterns, when to use which, and practice design pattern katas. One problem you will run into is with examples. For some reason, many examples are much too general and unrealistic. This is a shame. You end up having to synthesize information you read with the example given and the realities you know. If you are not very experienced, these examples might even feel pointless. Don't fear. Practice helps and we are here to help too.

One thing you will notice in common with many of the design patterns is the use of "has-a" rather than "is-a" object relationships. A request object *has a* mediator object. A metrics object *has a* mediator object. This is a powerful concept worth exploration and practice. Go for it.

See the "Resources" section for more information on design patterns.

Is this Pythonic?

Python is an interpreted language, so some of the advantages of these principles aren't relevant. Specifically, source code dependencies do not mean we have to suffer recompiling or building many modules when we could have used dependency inversion to only rebuild one. However, that isn't the only benefit of these principles. It is just a massive one when you are working in huge complex software projects that need many teams working on distinct parts of the system.

We still get the benefits of a more intuitive structure, loose coupling, and more freedom to solve problems that only appear after our software is in use by customers. Let's say we release a web application and find that we really need to switch to a horizontal scaling database like OrientDB. Oh no! We developed our software using Django and adding OrientDB support would take a long time. We are victims of our own success. To make it worse, we followed "web MVC" advice and put all kinds of business logic in the model! Woe to the fat model! Woe to tight coupling!

All we needed to do to protect ourselves was to separate and isolate our core business logic from all other concerns and dependencies. Then create an interface for PyOrient OGM. That's a much smaller lift that figuring out how to extend the Django ORM to work with OrientDB. We've investigated it. Which should tell you that some of us have made mistakes like this and learned from it.

Python also gives you incredible power to solve problems using polymorphism. We have multiple inheritance. We have functions that are objects. We can override fundamental behaviors of Python. That's cool. Some might say Python trusts us too much, but that means we need to take care to be responsible professionals.

Within the best of our knowledge and skills, we commit to do no harm to function or structure. And the maintainers can help you make the same commitment.

Design Red Flags

Fragility: Your change is causing tests to fail in modules you didn't change. Or Refactoring causes many tests to fail.

Rigidity: To add a new little feature, you must change code in many places.

Train Wrecks: Object.object.object.method().method().choo_choo

Tight Coupling: Logic for real world processes occupying the same space as or depending directly on frameworks, databases, drivers, and other I/O modules.

7.6 Resources

7.6.1 Awesome Authors:

Robert C. Martin (Uncle Bob): One of the original signers of the Agile Manifesto. *The* Clean Coder.

Martin Fowler: One of the original signers of the Agile Manifesto. Author of a fantastic book on refactoring.

Kent Beck: One of the original signers of the Agile Manifesto, creator of Extreme Programming, pioneer of design patterns, and test-driven development.

7.6.2 Testing

Google: Just Say No to More End-to-End Tests

Kinds of Tests: Uncle Bob – First Class Tests

7.6.3 Test-Driven Development:

Uncle Bob: The Three Laws of TDD (video)

Uncle Bob: The Pragmatics of TDD

Uncle Bob: Giving Up on TDD

Martin Fowler: Mocks Aren't Stubs

7.6.4 Object-Oriented Design

SOLID Principles

Design Principles and Design Patterns

Design Patterns on Wikipedia: This is a nice summary with examples.

Refactoring.Guru

Source Making: Design Patterns: This is a great resource for choosing from competing patterns and refactoring techniques. However, the Python examples aren't useful to many novices.

Uncle Bob: The Clean Architecture

7.6.5 Documentation

Getting Started with Sphinx

reStructuredText Primer

An Introduction to Sphinx and Read the Docs for Technical Writers

CHAPTER 8

Code of Conduct

8.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

8.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

8.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

8.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

8.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at oss-code-of-conduct@bigfishgames.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

8.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

CHAPTER 9

Indices and tables

- genindex
- modindex
- search