Developer Playbook

Introduction

The purpose of this document is to give all the information to the new development team member.

This document contains important information for the development team, tools that teams are using, plugins for IDEs, how to do estimations, how to work with Git, and how to deploy.

This document also contains best practices grouped by different languages.

Meet your team

As you embark on your onboarding journey, the first step involves familiarizing yourself with your project team. Your manager will initiate contact to facilitate introductions and help you get acquainted with your fellow team members.

During the initial phase, it's crucial to seek answers to the following questions::

- Who is your lead?
- Gain clarity on the roles of key team members: get contacts of DEV/BA Leads
- Who is a Project Manager from the ROKO's side?
- Who is a client? Understand the identity of the client you'll be collaborating with. Learn about the types of questions or challenges the client contact can assist in resolving, leveraging their expertise.

Development

Tools for development

- **.NET Development**
  - Prerequisite: Install .NET (version depends on project requirements)
  - Windows
    - IDE: Visual Studio
    - Plugins: CodeMaid, Trailing Whitespace VisualTrailing Whitespace Visualizerizerer, RoslynRoslynatorator, Visual Studio Spell Checker, Editor Editor Guidelines, Markdown Editor, File Icons
  - Cross-platform
    - IDE: Visual Studio Code (VS Code) or JetBrains Rider
    - Plugins: OmniSharp, OmniSharp, C# for Visual Studio Code
- **PHP**
  - Prerequisite: Install PHP (version depends on project requirements)
  - IDE**:** PHPStorm, Visual Studio Code (VS Code), NetBeans
- **Python**
  - Prerequisite: Install Python (version depends on project requirements)
  - IDE: Visual Studio Code (VS Code), JetBrains PyCharm
  - Plugins: Python
- **JavaScript**
  - Prerequisite: Install Node.js (version depends on project requirements)
  - IDE: Visual Studio Code (VS Code)
  - Plugins: ESLint, Prettier
- **Database**
  - **PostgreSQL**
    - Prerequisite: Install PostgreSQL Server (on-premises)
    - IDE: pgAdmin, DBeaver Community
  - **MySQL**
    - Prerequisite: Install MySQL Server (on-premises)
    - IDE: MySQL Workbench, phpMyAdmin, DBeaver Community, BeeKeeper Studio
  - **SQL Server**
    - Windows
      - Prerequisite: Install SQL Server (on-premises)
      - IDE: SQL Server Management Studio (SSMS), Azure Data Studio, DBeaver Community
    - Cross-platform
      - Prerequisite: Run SQL Server containerized (e.g., Docker)
      - IDE: Azure Data Studio, DBeaver Community

Engineering

Estimations

- Developers will use a Fibonacci-based points system to estimate all tickets.
  - 1,2,3,5,8,13
    - 1 point half day,
    - 2 points 1-2 days
    - 3 points 2-3 days,
    - 5 points 3-5 days
    - 8 points 1 dev sprint
    - 13 points divide in two sprints
  - Anything that is more complex than 5 points should be broken down to multiple tickets organized in user story.
  - Any ticket with 3, or more story points should be broken down in Tasks.

Sprint schedule

Processes

- Story-points must be assigned prior to sprint-planning.

All questions about fundamental requirements must be asked prior to the middle of development:

- If requirements need to be changed per dev feedback, developer needs to sign off on the updated logic/requirements.
- All tickets pulled into a sprint must have assigned story points.
- No mid-sprint Production deploys unless it is a blocker.
  - If so, must file as JIRA ticket and label as 'regression.'
- All hotfixes require post-mortem with the product, developers, and QA team-leads.
- 15-min post-sprint retro after every sprint
  - In this past sprint, did we do well and should continue doing? What did we not do well and should stop doing? What can we improve
- Communication on status of deploys via email for each environment:
  - Deployed to QA – **started** | **complete** - release notes via email.
  - Deployed to staging – **started** | **complete** - release notes via email.
  - Deployed to production – **started** | **complete** - release notes.
- Engineering lead will hold engineering team accountable to adhering to the sprint schedule, deliverables, and items in this document.

Unit Testing

All codebases should be covered by unit tests. Unit testing is a critical practice in software development for several important reasons:

- **Early Detection of Bugs**:
  - Unit tests allow developers to catch and address bugs and issues in the code at an early stage of development, reducing the cost and effort required to fix them later in the development cycle. This leads to more efficient and cost-effective software development.
- **Improved Code Quality**:
  - Writing unit tests encourage developers to write cleaner, more modular, and maintainable code. When you write unit tests, you naturally design your code to be more modular, making it easier to understand, modify, and extend.
- **Documentation**:
  - Unit tests serve as a form of documentation. They provide insights into the expected behavior of code and can help other developers understand how a particular component is supposed to work. This is especially valuable in complex projects with many contributors.
- **Regression Testing**:
  - When you make changes or add new features to your code, you can rerun the unit tests to ensure that existing functionality remains intact. This helps prevent the introduction of new bugs while modifying or extending the codebase.
- **Maintainability**:
  - Unit tests make it easier to maintain and refactor code. When you change code to add new features or improve performance, you can run the associated unit tests to check that you haven't inadvertently broken existing functionality.
- **Facilitates Collaboration**:
  - Unit testing promotes collaboration among developers. Team members can confidently work on different parts of the codebase without worrying that their changes will break existing functionality if unit tests are in place.
- **Continuous Integration**:
  - In a continuous integration (CI) environment, unit tests can be automatically run whenever code changes are pushed to a version control system. This ensures that any issues are quickly identified, and the integration of code from different team members is smooth.
- **Reduces Debugging Time**:
  - Unit tests pinpoint the location of issues, making it faster and more efficient to diagnose and fix problems. This contrasts with not having unit tests, where debugging might involve searching through the entire codebase.
- **Confidence in Code Changes**:
  - Unit tests give developers confidence that their code is working as expected. This confidence allows for more frequent and smaller code changes, reducing the risk of making large, error-prone changes.
- **Test-Driven Development (TDD)**:
  - Unit testing is a core practice in Test-Driven Development, a methodology where tests are written before code is implemented. TDD helps ensure that code meets the specified requirements, and that the software is designed with testability in mind.

In summary, unit testing is essential for ensuring software quality, reducing development time and costs, and fostering collaboration among development teams. It is a key practice in modern software development that contributes to the creation of reliable, maintainable, and efficient software systems.

Repository

Brancing Strategy

Branching strategy is a fundamental aspect of version control in software development, helping teams manage code changes, collaborate effectively, and maintain a stable and organized codebase. There are several branching strategies, each with its own purpose and best practices. One of the most used strategies is the Gitflow branching model. Here's an overview:

Gitflow Branching Model:

The Gitflow branching model is a branching strategy designed to support feature development, bug fixing, and release management in a structured and organized manner.

Key branches in Gitflow include:

- **Master**: The **master** branch represents the stable production-ready code. It should always contain code that's considered "live" or in a production environment. Directly deploying to **master** is typically reserved for releases.
- **Develop**: The **develop** branch is where ongoing development work happens. New features, bug fixes, and other changes are integrated into this branch. It's considered a somewhat stable branch but may contain work in progress.
- **Feature branches**: These branches are created for developing new features or enhancements. Each feature or task gets its dedicated branch. Feature branches are usually branched off **develop** and merged back into it upon completion.
- **Release branches**: When the development in the **develop** branch reaches a state where a new release is imminent, a release branch is created. This branch allows for final testing, bug fixes, and preparation for a release.
- **Hotfix branches**: Hotfix branches are used to quickly address critical issues in the production code. They branch off **master**, get the necessary fixes, and are merged back into both **master** and **develop**.

The typical flow in Gitflow is as follows:

- Developers work on feature branches derived from **develop**.
- When a feature is complete, it's merged back into **develop**.
- When it's time for a release, a **release** branch is created from **develop**.
- After final testing and adjustments in the **release** branch, it's merged into **master** and **develop**.
- Hotfix branches are created from **master** to address critical issues in production.

Advantages of the Gitflow branching model:

- **Organization**: It provides a clear structure for development, release, and hotfix phases.
- **Isolation**: Feature branches keep the main development branch (**develop**) relatively stable.
- **Versioning**: It facilitates versioning by explicitly defining where each codebase comes from and goes to.
- **Team Collaboration**: It allows multiple team members to work on different features concurrently without interfering with each other's work.

- **Stability**: The **master** branch remains stable since only tested and approved code is merged into it.

- One branch per card
- Branches types:
  - Master development branch
  - QA
  - Stage (UAT is done here)
  - Production
  - Feature branches

Tagging Process in GIT

In Git, tagging is the process of associating a meaningful name or label with a specific commit or a range of commits. Tags serve as references to specific points in Git history, such as release points or significant milestones. Tags are typically used to mark versions, releases, or notable points in project's development history. There are two main types of tags in Git: lightweight tags and annotated tags.

Lightweight Tags

Lightweight tags are simple labels associated with a specific commit. They do not contain any additional information like the tagger's name, email, or a message. To create a lightweight tag, you use the following command:

git tag tag_name commit_id

- '**tag_name**' is the name of the tag you want to create.

- '**commit_id**' is the SHA-1 hash of the commit you want to tag.

Example:

git tag v1.0.0 abc1234

Annotated Tags

Annotated tags, unlike lightweight tags, store additional information such as the tagger's name, email, date, and an optional message. They are useful for providing more context about the tagged commit. To create an annotated tag, you use the **'-a'** or **'—annotate'** option with the **'git tag'** command:

git tag -a tag_name -m "Tag message" commit_id

- **'tag_name'** is the name of the tag.

- **'"Tag message"'** is an optional message describing the tag.

- **'commit_id'** is the SHA-1 hash of the commit you want to tag.

Example:

git tag -a v1.0.0 -m "Initial release" abc1234

1. **Tagging Best Practices**
2. **Use Annotated Tags for Releases:** Annotated tags are recommended for marking releases and significant points in project's history because they provide more context and information.
3. **Consistent Naming Conventions:** Establish a clear and consistent naming convention for tags. This helps quickly identify the purpose of each tag, such as version numbers (e.g., 'v1.0.0') or release names.
4. **Use Tags for Stable Points:** Tags are typically used to mark stable and well-tested points in project's history. Avoid tagging experimental or incomplete work.
5. After creating tags, push them to a remote repository using the 'git push' command. This allows collaborators to access and use the tags.

To push all tags to a remote repository, use the following command:

git push --tags

- **List Tags:**

  To list the available tags in repository, use the **'git tag'** command:

  git tag

- **Checkout a Tag:**

  To check out a specific tag to create a detached HEAD state. This can be useful for inspecting the code at a specific point in history or for creating a new branch based on a tag:

  git checkout tag_name

Tags are essential for maintaining a clear version history and marking important milestones in Git project. They make it easy to reference specific commits and provide context about releases and significant changes.

- Each ticket in JIRA is created in a branch.
- Once the ticket is implemented, a pull request is created, and the JIRA ticket is moved to In Review Status
- Pull Request is then reviewed. If approved, it will be merged into "Master Branch" (for dev environment)
- Finally, after the dev lead reviews the implemented functionality, JIRA ticket will be moved to "Resolved" status.
- Tags are used to mark commits to show sprints, versions. These can be used to switch between branches.

A screenshot of a computer Description automatically generated

---

Pull Requests

A pull request (PR), also known as a merge request in some version control systems like GitLab, is a fundamental component of collaborative software development. It's a mechanism for proposing, reviewing, and discussing code changes before they are merged into the main codebase.

Key Aspects of Pull Requests:

- **Creation**:
  - A developer creates a pull request when they want to merge a set of changes from a feature branch (or any branch) into a target branch, typically the main branch (e.g., **master**, or **main**).
- **Code Review**:
  - The pull request provides an opportunity for peer review. Other developers, or a team lead, can review the proposed changes, offering feedback and suggestions for improvement. Code review helps ensure code quality, maintainability, and adherence to coding standards.
- **Discussion**:
  - Pull requests include a comment section where discussions about the code changes take place. Developers can ask questions, provide feedback, and discuss any concerns related to the changes.
- **Continuous Integration (CI) Integration**:
  - Many platforms that support pull requests also integrate with CI tools. CI systems run automated tests to validate the code changes, ensuring that they don't introduce new issues.
- **Status Checks**:
  - Pull requests can be configured to pass certain criteria, such as successful test runs and code analysis before they can be merged.
- **Iterative Development**:
  - Developers can make additional commits to the pull request based on feedback received during the review process. These new commits are automatically included in the pull request, allowing for an iterative development process.
- **Approval**:
  - Pull requests often require one or more approvals from designated reviewers before they can be merged. This ensures that code is thoroughly examined, and consensus is reached.
- **Merging**:
  - Once the code changes in the pull request are reviewed and approved, they can be merged into the target branch. This action typically closes the pull request.

How do pull requests works.

1. Create a branch for ticket.
2. Create pull request from that branch.
3. Pull request is reviewed by other team members.
4. Pull request is merged into master.
5. Related ticket is marked Review.

Code review for quality before merging PR The developer reviews the code for quality before merging in PR. The dev lead then reviews related tickets using dev environment website.

1. If everything checks out the ticket is marked Review Passed
2. If there are issues the dev lead Re-Opens the ticket, it goes back to To-Do.
3. The expectation is that the developer will check changes into #2 from above and continue from there.

Deployment

Deploying to QA Environment

- Dev Lead creates new builds on QA/Stage Environments
  - All tickets that have been "Resolved" should be moved to "QA Ready". Once in a QA environment the ticket can be tested.
- If a ticket is in In Review status but has already been merged in into the dev environment – do not create new build.
- PM/Business must decide when a ticket is moved to Resolve status. If a ticket is in the QA environment with a status of "In Review" this rule has been violated.
- Do not deploy to Stage before all tickets in QA environment are moved to QA Ready status.

Deploying to Stage Environment

- Once all tickets are verified, explicit QA approval is required before deploying to the Stage environment. Do NOT deploy to stage without QA approval.
- To create a new build on Stage all tickets must be in 'Verified" status.
- QA may provide a list of issues that are not closed, and the team will determine if the issues are critical or not.
  - If the issues are critical, they must be fixed prior to going to stage.
- If the issue is not critical, deployment to stage is allowed, followed by a hot fix after build or leave as is.

Code Versioning

The goal of Code Versioning is to create a workflow that will give a clear visibility into which tickets are associated with releases and when they will be released to production.
Versions: A milestone grouping of tickets.

- **Release**: Version(s) that will be deployed to production on a specific date.

Process for Associating Tickets to Versions

Versions will be used to track different groupings of tickets through the different environments (QA, Stage, Prod) as we move through the product development cycle. The version description will identify the environment on which the tickets are deployed.
Tickets will be associated to versions once they have gone through a full QA cycle and our QA team determines that ticket is ready to be included in the upcoming release. All tickets that are in the Verified Column of our boards must have a version associated with it. The QA team will tag the tickets to the upcoming version, as they move them from In Review to Verified status.
Versioning will be particularly useful to track tickets from different sprints and epics that are deployed to the same environment simultaneously. It will also provide a clear picture of the upcoming product releases by listing all tickets that are being prepared for deployment.
To see all tickets in a particular Version along with information about the Sprint in which each ticket was completed, go to the Backlog section in Jira, click on the Versions tab and select a Version. Only the tickets associated with that version will be displayed. You may need to expand each Sprint individually to see tickets for that version in the sprint. Expand the Version details to see the total number of Issues in that Version, and how many are in a Completed status.

A screenshot of a computer Description automatically generated

To see a list of issues in a Version, click on Releases in the left menu and click on the version number in the Version column.

A screenshot of a computer Description automatically generated

---

Release Management

There will be a separate Release action for each Version that is going to Production. Where possible, we will set a target date for the release while still on Stage. The description of the release will indicate the current environment of the tickets. We will update the description as the Versions go through the different environments prior to being released to Production. Once we have all tickets associated with the release version and the release takes place, we will take the "Release" action in this tab.
Each released version can be drilled down to reveal all tickets associated with that release. Any upcoming releases that have tickets already associated to them can be viewed here to provide visibility of what is being built included in the next release.

A screenshot of a computer Description automatically generated

---

Release Naming Conventions

Standard releases that occur at the end of a sprint(s), the version and release name will be counted as a whole numerical addition, V1, V2, V3...
For any hot-fix or minimal release, this will be counted as a dot release, 1.1, 1.2,1.3....

Tracking Historical Releases

The Releases section in Jira contains information for all historical releases, including a breakdown by tickets per release. This view also displays the date of the release to production.

A screenshot of a computer Description automatically generated

---

Best practices

Coding naming conventions

C#

Coding naming conventions in C# are a set of rules and guidelines that help developers choose meaningful and consistent names for various code elements, such as variables, methods, classes, and more. Consistent naming conventions make code more readable, maintainable, and understandable, benefiting both the original developers and those who may work with or maintain the code in the future. Here are some common naming conventions in C#:

- **Camel Case:**
  - Variables and fields: 'myVariable', 'customerName', 'itemCount'
  - Parameters: 'paramValue', 'inputData'
  - Local variables: 'count', 'resultValue'
- **Pascal Case:**
  - Class names: 'Person', 'OrderManager', 'UserSettings'
  - Method names: 'CalculateTotal', 'SaveData', 'InitializeUser'
  - Property names: 'FirstName', 'ProductPrice', 'IsEnabled'
- **Uppercase for Constants:**
  - Constants: 'MAX_LENGTH', 'PI', 'DEFAULT_TIMEOUT'
- **Abbreviations:**
  - Avoid excessive use of abbreviations, but if used, follow common abbreviations, such as 'ID' for "identifier" or 'URL' for "uniform resource locator."
- **Verb-Noun for Methods:**
  - Name methods using a verb or action followed by a noun or object, indicating what the method does. For example, 'CalculateTotalPrice()', 'LoadDataFromFile()', or 'CreateNewUser()'.
- **Plural for Collections:**
  - Use plural names for collections and arrays to indicate that they contain multiple items, e.g., 'customers', 'orders', 'products'.
- **Boolean Variables:**
  - Name boolean variables as questions or states to make code more readable. For example, 'isActive', 'hasPermission', or 'isAvailable'.
- **Avoid Hungarian Notation:**
  - Avoid using Hungarian notation, where variable names include data type prefixes like 'strName' or 'intCount'. Modern IDEs provide type information, making these prefixes unnecessary.
- **Meaningful Names:**
  - Choose names that are descriptive and convey the purpose or intent of the code element. Avoid vague or generic names like 'data', 'temp', or 'value'.
- **Namespaces:**
  - Use meaningful namespaces to organize related classes, e.g., 'MyApp.Data', 'MyApp.Utilities', or 'System.IO'.
- **Event Handlers:**
  - Use the '(object sender, EventArgs e)' signature for event handler methods.
- **Acronyms:**
  - For acronyms, use camel case or Pascal case depending on the context. For example, 'XMLParser' (Pascal case) or 'xmlData' (camel case).
- **Don't Clash with Keywords:**
  - Avoid using C# keywords or reserved words as identifiers. For example, don't name a variable "namespace" or "class."

It's essential to follow the naming conventions consistently within your codebase and adhere to any established conventions within your development team or organization. This consistency enhances code readability and maintainability and makes collaboration more efficient. Additionally, tools like ReSharper and Visual Studio can help enforce and check naming conventions.

Python

In Python, coding naming conventions are a set of rules and guidelines that help developers choose meaningful and consistent names for various code elements, including variables, functions, classes, and modules. Adhering to naming conventions in Python enhances code readability, maintainability, and collaboration. The most followed naming conventions in Python are outlined in the PEP 8 style guide, which is considered the authoritative guide for Python code style. Here are some key Python naming conventions according to PEP 8:

- **Variable Names:**
  - Use lowercase letters and separate words with underscores (snake_case). For example: my_variable, count, total_price.
- **Constant Names:**
  - Use uppercase letters and separate words with underscores. Constants are typically defined at the module level. For example: MAX_VALUE, PI, DEFAULT_TIMEOUT.
- **Function and Method Names:**
  - Use lowercase letters and separate words with underscores (snake_case). For example: calculate_total_price(), process_data(), get_user_name().
- **Class Names:**
  - Use CapitalizedWords (PascalCase). Class names should start with an uppercase letter, and words are concatenated without underscores. For example: MyClass, EmployeeDetails, HTTPRequestHandler.
- **Module Names:**
  - Use lowercase letters and separate words with underscores. Modules should have short, lowercase names. For example: my_module, data_processing, utils.
- **Package Names:**
  - Use lowercase letters and avoid underscores. Package names should be short and descriptive. For example: mypackage, webtools, api.
- **Instance Variables:**
  - Use the same conventions as variable names (lowercase with underscores). For example: self.my_var, self.total_count.
- **Function Arguments:**
  - Use lowercase letters and separate words with underscores. For example: user_id, input_data, file_path.
- **Method Arguments and Local Variables:**
  - Use lowercase letters and separate words with underscores. For example: self.user_id, result_value, file_handle.
- **Avoid Confusing Names:**
  - Avoid using names that are like built-in Python functions or reserved words, which can lead to confusion.
- **Constants within a Module:**
  - If you need module-level constants, use uppercase names. For example: MODULE_CONSTANT.
- **Acronyms:**
  - Acronyms should be treated as words in names, and the case should be consistent. For example: XMLParser, httpRequest, htmlDecoder.
- **Public and Private Names:**
  - By convention, names starting with a single underscore (e.g., _private_var) are considered non-public and are intended for internal use within the module. Names starting with a double underscore (e.g., __private_var) are name-mangled to make them less likely to accidentally conflict with names in subclasses.
- **Naming Conventions for Special Methods:**
  - Special methods (also known as "magic methods") in Python, like __init__ and __str__, have specific naming conventions. These names should not be used for your custom methods or attributes.

It's essential to adhere to these naming conventions to ensure code consistency and readability. Following PEP 8 guidelines makes your code more accessible and understandable to other Python developers, and it's considered a best practice in the Python community. Many Python code editors and integrated development environments (IDEs) can help you automatically format your code according to these conventions.

Node.js

Coding naming conventions in Node.js, like in any JavaScript-based development, are essential for writing clean, consistent, and maintainable code. Node.js is often used for building server-side applications and APIs, and adhering to naming conventions can make your code more understandable and reduce potential issues. While there isn't a single authoritative style guide for Node.js like PEP 8 for Python or PEP 20 for CPython, many Node.js developers follow common JavaScript naming conventions such as those outlined in the Airbnb JavaScript Style Guide and the Google JavaScript Style Guide. Here are some general naming conventions for Node.js development:

- **Variable and Function Names:**
  - Use camelCase for variable and function names. For example: myVariable, calculateTotalPrice(), getUserInfo().
- **Constant Names:**
  - Use UPPERCASE_SNAKE_CASE for constants. For example: MAX_VALUE, API_KEY, DEFAULT_PORT.
- **Class Names:**
  - Use PascalCase (also known as CamelCase) for class names. For example: MyClass, UserController, HttpServer.
- **File Names:**
  - Use kebab-case or snake_case for file names. For example: my-module.js, utility_functions.js, config_settings.json.
- **Module and Package Names:**
  - Use kebab-case or snake_case for module and package names, similar to file names. For example: my-module, data-processing, api_v2.
- **Folder Names:**
  - Use lowercase and separate words with underscores or hyphens for folder names. For example: my_project, user-data, public_assets.
- **Variable Prefixes:**
  - Avoid using Hungarian notation or prefixing variables with their data types. For example, don't use strName for string variables or nCount for number variables.
- **Acronyms:**
  - Treat acronyms and abbreviations as words. Use consistent capitalization for acronyms. For example: XMLParser, HTTPRequest, userID.
- **Function and Method Naming:**
  - Choose descriptive names that reflect the function's purpose. Use verbs or action-oriented names for functions. For example: calculateTotalPrice(), getUserData(), sendHTTPRequest().
- **Callback Functions:**
  - If you're using callbacks, make sure the callback function name is meaningful and describes the operation. For example: fs.readFile('file.txt', (err, data) => {...}).
- **Avoid Reserved Keywords:**
  - Avoid using JavaScript reserved words and keywords as identifiers. This includes words like function, class, return, and so on.
- **Consistency:**
  - The most important aspect of naming conventions is consistency. It's essential to follow the same conventions throughout your codebase and ensure that your team members do the same.

While there isn't a single official standard for Node.js naming conventions, it's important to choose a style guide or set of conventions that you and your team are comfortable with and consistently apply them throughout your Node.js project. Adhering to these conventions will help make your code more maintainable and readable, which is particularly valuable in large and collaborative projects.

PHP

Coding naming conventions in PHP are a set of guidelines and rules that developers follow to name variables, functions, classes, and other elements in their PHP code. Consistent naming conventions make code more readable and maintainable, which is essential for collaborative development and codebase longevity. While there isn't a single, universally accepted naming convention for PHP, there are some widely followed practices. Here are some common PHP naming conventions:

- **Class Names:**
  - Use PascalCase (also known as CamelCase) for class names.
  - Start each word with an uppercase letter.
  - Example: MyClass, DatabaseConnection, UserController
- **Function and Method Names:**
  - Use camelCase for function and method names.
  - Start with a lowercase letter, and each subsequent word begins with an uppercase letter.
  - Example: getUserById(), calculateTotalPrice(), validateEmail()
- **Variable Names:**
  - Use camelCase for variable names as well.
  - Start with a lowercase letter, and each subsequent word begins with an uppercase letter.
  - Example: $userName, $totalAmount, $userList
- **Constant Names:**
  - Use uppercase letters with underscores to separate words.
  - Example: MAX_LENGTH, DB_HOST, API_KEY
- **Namespace Names:**
  - Use camelCase or PascalCase for namespaces.
  - Follow the same conventions as class names.
  - Example: MyNamespace\SubNamespace, Vendor\Package
- **File Names:**
  - Use lowercase letters and underscores for file names.
  - Typically, use the same name as the class contained within, and add the .php extension.
  - Example: user_model.php, database_connection.php
- **Database Table and Column Names:**
  - Use lowercase letters and underscores for table and column names.
  - Tables and columns should be named in a way that reflects their content and purpose.
  - Example: users, user_id, created_at
- **Acronyms and Abbreviations:**
  - Acronyms and abbreviations should follow the same capitalization rules as other words. For example, use getUserInfo() instead of getUserInfo().
- **Singular vs. Plural:**
  - For class names, use the singular form if the class represents a single entity (e.g., User for a user entity).
  - For variable names, use the plural form if the variable holds an array or collection of items (e.g., $users for an array of user objects).
- **Consistency:**
  - The most important aspect of naming conventions is consistency. Pick a convention and stick with it throughout your codebase.

Remember that while these conventions are widely followed, it's essential to follow the conventions of the project you are working on or the coding standards of your team or organization. Consistency within a codebase is more important than any specific convention.

Additional resources

- Automated Release Notes
- SonarQube Playbook
- Jira
- Git

1 person liked this
129 Views
Save for later