**PROJECT GUIDELINES**

🙞 🕮 🙜

1. **Git** 
   1. Some Git rules
   2. Git workflow
   3. Writing good commit messages
2. **Documentation**
3. **Environments** 
   1. Consistent dev environments
   2. Consistent dependencies
4. **Dependencies**
5. **Testing**
6. **Structure and Naming**
7. **Code style**
8. **Logging**
9. **API** 
   1. API Design
   2. API security
   3. API documentation

**1. Git**

**1.1 Some Git rules**

|  |  |
| --- | --- |
| **Rules to keep in mind** | **Why** |
| * Perform work in a **feature branch.** | Because this way ***all work is done in isolation on a dedicated branch rather than the main branch.*** It allows you to submit multiple pull requests without confusion. You can iterate without polluting the master branch with potentially unstable, unfinished code. [**Read more...**](https://www.atlassian.com/git/tutorials/comparing-workflows#feature-branch-workflow) |
| * **Branch out from develop** | This way, you can make sure that ***code in master will almost always build without problems, and can be mostly used directly for releases*** (this might be overkill for some projects) |
| * Never push into develop or master branch. **Make a Pull Request** | It notifies team members that they have completed a feature. It also enables easy peer-review of the code and dedicates forum for discussing the proposed feature |
| * Update your local develop branch and do an interactive rebase before pushing your feature and making a Pull Request | Rebasing will merge in the requested branch (master or develop) and apply the commits that you have made locally to the top of the history without creating a merge commit (assuming there were no conflicts). Resulting in a nice and clean history. [**Read more ...**](https://www.atlassian.com/git/tutorials/merging-vs-rebasing) |
| * **Resolve potential conflicts** **while rebasing** and **before** making a **Pull Request** * **Delete local and remote feature branches after merging** | It will clutter up your list of branches with dead branches. It insures you only ever merge the branch back into (master or develop) once. ***Feature branches should only exist while the work is still in progress.*** |
| * Before making a **Pull Request**, make sure your feature branch builds successfully and passes all tests (including code style checks). | You are about to add your code to a stable branch. If your feature-branch tests fail, there is a high chance that your destination branch build will fail too. Additionally you need to apply code style check before making a Pull Request. It aids readability and reduces the chance of formatting fixes being mingled in with actual changes. |
| * Use this **.gitignore** file. | It already has a list of system files that should not be sent with your code into remote repository. In addition, it excludes setting folders and files for mostly used editors, as well as most common dependency folders. |
| * Protect your develop and master branch | It protects your production-ready branches from receiving unexpected and irreversible changes. [**Read more**...](https://help.github.com/articles/about-protected-branches/) Github and Bitbucket |

**1.2 Git workflow**

Because of most of the reasons above, we use [Feature-branch-workflow](https://www.atlassian.com/git/tutorials/comparing-workflows#feature-branch-workflow) with [Interactive Rebasing](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#the-golden-rule-of-rebasing) and some elements of [Git flow](https://www.atlassian.com/git/tutorials/comparing-workflows#gitflow-workflow) (naming and having a develop branch). The main steps are as follow:

|  |  |  |
| --- | --- | --- |
| **What** | **How** | **Why** |
| Checkout a new feature/bug-fix branch | git checkout -b <branchname> |  |
| Make Changes | git add  git commit –a | git commit -a will start an editor which lets your separate the subject from the body. Read more about it in section 1.3. |
| Sync with remote to get changes you’ve missed | git checkout develop  git pull | This will give you a chance to deal with conflicts on your machine while rebasing (later) rather than creating a Pull Request that contains conflicts. |
| Update your feature branch with latest changes from develop by interactive rebase | git checkout <branchname>  git rebase -i --autosquash develop | You can use --autosquash to squash all your commits to a single commit. Nobody wants many commits for a single feature in develop branch [**Read more...**](https://robots.thoughtbot.com/autosquashing-git-commits) |
| If you don’t have conflict, skip this step. If you have conflicts, [resolve them](https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/) and continue rebase | git add <file1> <file2> ...  git rebase –continue |  |
| Push your branch. Rebase will change history, so you'll have to use -f to force changes into the remote branch. If someone else is working on your branch, use the less destructive --force-with-lease. | git push –f | When you do a rebase, you are changing the history on your feature branch. As a result, Git will reject normal git push. Instead, you'll need to use the -f or --force flag. [**Read more...**](https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/) |
| Make a Pull Request. |  |  |
| Pull request will be accepted, merged and close by reviewer. |  |  |
| Remove your local feature branch if you're done. |  |  |

**1.3 Writing good commit messages**

Having a good guideline for creating commits and sticking to it makes working with Git and collaborating with others a lot easier. Here are some rules of thumb ([source](https://chris.beams.io/posts/git-commit/#seven-rules)):

|  |  |
| --- | --- |
| **What** | **Why** |
| Separate the subject from the body with a newline between the two | Git is smart enough to distinguish the first line of your commit message as your summary. In fact, if you try git shortlog, instead of git log, you will see a long list of commit messages, consisting of the id of the commit, and the summary only |
| Limit the subject line to 50 characters and Wrap the body at 72 characters | Commits should be as fine-grained and focused as possible, it is not the place to be verbose. [**Read more...**](https://medium.com/@preslavrachev/what-s-with-the-50-72-rule-8a906f61f09c) |
| Capitalize the subject line |  |
| Do not end the subject line with a period |  |
| Use [imperative mood](https://en.wikipedia.org/wiki/Imperative_mood) in the subject line | Rather than writing messages that say what a committer has done. It's better to consider these messages as the instructions for what is going to be done after the commit is applied on the repository. [**Read more...**](https://news.ycombinator.com/item?id=2079612) |
| Use the body to explain what and why as opposed to how |  |

**2. Documentation**

* Use this template for [**README.md**](https://github.com/wearehive/project-guidelines/blob/master/README.sample.md), Feel free to add uncovered sections.
* For projects with more than one repository, provide links to them in their respective README.md files.
* Keep README.md updated as project evolves.
* Comment your code. Try to make it as clear as possible what you are intending with each major section.
* If there is an open discussion on github or stackoverflow about the code or approach you're using, include the link in your comment,
* Don't use commenting as an excuse for a bad code. Keep your code clean.
* Don't use clean code as an excuse to not comment at all.
* Keep comments relevant as your code evolves.

**3. Environments**

|  |  |
| --- | --- |
| **What** | **Why** |
| Define separate development, test and production environments if needed. | Different data, tokens, APIs, ports etc... might be needed on different environments. You may want an isolated development mode that calls fake API which returns predictable data, making both automated and manually testing much easier. Or you may want to enable Google Analytics only on production and so on. [**Read more...**](https://stackoverflow.com/questions/8332333/node-js-setting-up-environment-specific-configs-to-be-used-with-everyauth) |
| Load your deployment specific configurations from environment variables and never add them to the codebase as constants, [look at this sample.](https://github.com/wearehive/project-guidelines/blob/master/config.sample.js) | You have tokens, passwords and other valuable information in there. Your config should be correctly separated from the app internals as if the codebase could be made public at any moment.  How  Use .env files to store your variables and add them to .gitignore to be excluded. Instead commit a .env.example which serves as a guide for developers. For production you should still set your environment variables in the standard way. [**Read more**](https://medium.com/@rafaelvidaurre/managing-environment-variables-in-node-js-2cb45a55195f) |
| It’s recommended to validate environment variables before your app starts. [Look at this sample](https://github.com/wearehive/project-guidelines/blob/master/configWithTest.sample.js) using joi to validate provided values. | It may save others from hours of troubleshooting. |

**3.1 Consistent dev environments:**

|  |  |
| --- | --- |
| **What** | **Why** |
| Set your node version in engines in package.json | It lets others know the version of node the project works on. [**Read more...**](https://docs.npmjs.com/files/package.json#engines) |
| Additionally, use nvm and create a .nvmrc in your project root. Don't forget to mention it in the documentation | Anyone who uses nvm can simply use nvm use to switch to the suitable node version. [**Read more...**](https://github.com/creationix/nvm) |
| It's a good idea to setup a preinstall script that checks node and npm versions | Some dependencies may fail when installed by newer versions of npm. |
| Use Docker image if you can. | It can give you a consistent environment across the entire workflow. Without much need to fiddle with dependencies or configs. [**Read more...**](https://hackernoon.com/how-to-dockerize-a-node-js-application-4fbab45a0c19) |
| Use local modules instead of using globally installed modules | Lets you share your tooling with your colleague instead of expecting them to have it globally on their systems. |

**3.2 Consistent dependencies:**

|  |  |
| --- | --- |
| **What** | **Why** |
| Make sure your team members get the exact same dependencies as you | Because you want the code to behave as expected and identical in any development machine[**. Read more...**](https://medium.com/@kentcdodds/why-semver-ranges-are-literally-the-worst-817cdcb09277)  How: Use package-lock.json on npm@5 or higher  I don't have npm@5:  Alternatively you can use Yarn and make sure to mention it in README.md . Your lock file and package.json should have the same versions after each dependency update. read more...  I don't like the name Yarn :  Too bad. For older versions of npm , use —save --save-exact when installing a new dependency and create npm-shrinkwrap.json before publishing. read more... |

**4. Dependencies**

|  |  |
| --- | --- |
| **What** | **Why** |
| Keep track of your currently available packages: e.g., **npm ls --depth=0** . [**Read more...**](https://docs.npmjs.com/cli/ls) |  |
| See if any of your packages have become unused or irrelevant: **depcheck** . [**Read more...**](https://www.npmjs.com/package/depcheck) | You may include an unused library in your code and increase the production bundle size. Find unused dependencies and get rid of them. |
| Before using a dependency, check its download statistics to see if it is heavily used by the community: **npm-stat** . [**Read more...**](https://npm-stat.com/) | More usage mostly means more contributors, which usually means better maintenance, and all of these result in quickly discovered bugs and quickly developed fixes. |
| Before using a dependency, check to see if it has a good, mature version release frequency with a large number of maintainers: e.g., **npm view async** . [**Read more...**](https://docs.npmjs.com/cli/view) | Having loads of contributors won’t be as effective, if maintainers don’t merge fixes and patches quickly enough. |
| If a less known dependency is needed, discuss it with the team before using it.  Always make sure your app works with the latest version of its dependencies without breaking: **npm outdated** . [**Read more...**](https://docs.npmjs.com/cli/outdated) | Dependency updates sometimes contain breaking changes. Always check their release notes when updates show up. Update your dependencies one by one, that makes troubleshooting easier if anything goes wrong. Use a cool tool such as **npm-check-updates.** |
| Check to see if the package has known security vulnerabilities with, e.g., [**Snyk**](https://snyk.io/test?utm_source=risingstack_blog). |  |

**5. Testing**

|  |  |
| --- | --- |
| **What** | **Why** |
| Have a **test** mode environment if needed. | While sometimes end to end testing in **production** mode might seem enough, there are some exceptions: One example is you may not want to enable analytical information on a 'production' mode and pollute someone's dashboard with test data. The other example is that your API may have rate limits in **production** and blocks your test calls after certain amount of requests. |
| Place your test files next to the tested modules using **\*.test.js** or **\*.spec.js** naming convention, like **moduleName.spec.js** | You don't want to dig through a folder structure to find a unit test. [**Read more...**](https://hackernoon.com/structure-your-javascript-code-for-testability-9bc93d9c72dc) |
| Put your additional test files into a separate test folder to avoid confusion. | Some test files don't particularly relate to any specific implementation file. You have to put it in a folder that is most likely to be found by other developers: **\_\_test\_\_** folder. This name: **\_\_test\_\_** is also standard now and gets picked up by most JavaScript testing frameworks. |
| Write testable code, avoid side effects, extract side effects, write pure functions | You want to test business logic as a separate unit. You have to "minimize the impact of randomness and non-deterministic processes on the reliability of your code". [**Read more...**](https://medium.com/javascript-scene/tdd-the-rite-way-53c9b46f45e3)  A pure function is a function that always returns the same output for the same input. Conversely, an impure function is one that may have side effects or depends on conditions from the outside to produce a value. That makes it less predictable. [**Read more...**](https://hackernoon.com/structure-your-javascript-code-for-testability-9bc93d9c72dc) |
| Use a static type checker | Sometimes you may need a Static type checker. It brings a certain level of reliability to your code. [**Read more...**](https://medium.freecodecamp.org/why-use-static-types-in-javascript-part-1-8382da1e0adb) |
| Run tests locally before making any pull requests to develop. | You don't want to be the one who caused production-ready branch build to fail. Run your tests after your rebase and before pushing your feature-branch to remote repository. |
| Document your tests including instructions in the relevant section of your README.md file. | It's a handy note you leave behind for other developers or DevOps experts or QA or anyone who gets lucky enough to work on your code. |

**6. Structure and Naming**

|  |  |
| --- | --- |
| **What** | **Why** |
| Organize your files around product features / pages / components, not roles. Also, place your test files next to their implementation. | Instead of a long list of files, you will create small modules that encapsulate one responsibility including its test and so on. It gets much easier to navigate through and things can be found at a glance. |
| **Bad** | **Good** |
| ├── controllers  | ├── product.js  | └── user.js  ├── models  | ├── product.js  | └── user.js | ├── product  | ├── index.js  | ├── product.js  | └── product.test.js  ├── user  | ├── index.js  | ├── user.js  | └── user.test.js |

|  |  |
| --- | --- |
| **What** | **Why** |
| Put your additional test files to a separate test folder to avoid confusion. | It is a time saver for other developers or DevOps experts in your team. |
| Use a **./config** folder and don't make different config files for different environments. | When you break down a config file for different purposes (database, API and so on); putting them in a folder with a very recognizable name such as **config** makes sense. Just remember not to make different config files for different environments. It doesn't scale cleanly, as more deploys of the app are created, new environment names are necessary. Values to be used in config files should provided by environment variables. [**Read more...**](https://medium.com/@fedorHK/no-config-b3f1171eecd5) |
| Put your scripts in a **./scripts** folder. This includes **bash** and **node** scripts. | It's very likely you may end up with more than one script, production build, development build, database feeders, database synchronization and so on. |
| Place your build output in a **./build** folder. Add **build/**  to **.gitignore.** | Name it what you like,  **dist** is also cool. But make sure that keep it consistent with your team. What gets in there is most likely generated (bundled, compiled, transpiled) or moved there. What you can generate, your teammates should be able to generate too, so there is no point committing them into your remote repository. Unless you specifically want to. |
| * Use **‘PascalCase’** **‘camelCase’**  for filenames and directory names. Use **PascalCase**  only for Components. * **CheckBox/index.js** should have the CheckBox component, as could **CheckBox.js**, but not **CheckBox/CheckBox.js** or **checkbox/CheckBox.js** which are redundant. * Ideally the directory name should match the name of the default export of **index.js**. | Then you can expect what component or module you will receive by simply just importing its parent folder. |

**7. Code style**

|  |  |
| --- | --- |
| **What** | **Why** |
| Use stage-2 and higher JavaScript (modern) syntax for new projects. For old project stay consistent with existing syntax unless you intend to modernize the project. | This is all up to you. We use transpilers to use advantages of new syntax. Stage-2 is more likely to eventually become part of the spec with only minor revisions. |
| Include code style check in your build process. | Breaking your build is one way of enforcing code style to your code. It prevents you from taking it less seriously. Do it for both client and server-side code. [**Read more...**](https://www.robinwieruch.de/react-eslint-webpack-babel/) |
| Use [ESLint - Pluggable JavaScript linter](http://eslint.org/) to enforce code style. | We simply prefer **eslint** , you don't have to. It has more number of rules supported, ability to configure the rules, and ability to add custom rules. |
| * We use [**Airbnb JavaScript Style Guide**](https://github.com/airbnb/javascript) for JavaScript, [**Read more**](https://www.gitbook.com/book/duk/airbnb-javascript-guidelines/details). Use the javascript style guide required by the project or your team. * We use [**Flow type style check rules for ESLint**](https://github.com/gajus/eslint-plugin-flowtype). when using [**Flow Type**](https://flow.org/). | Flow introduces few syntaxes that also need to follow certain code style and be checked. |
| Use .eslintignore to exclude file or folders from code style check. | You don't have to pollute your code with eslint-disable comments whenever you need to exclude couple of files from style checking. |
| Remove any of your eslint disable comments before making a Pull Request. | It's normal to disable style check while working on a code block to focus more on the logic. Just remember to remove those eslint-disable comments and follow the rules. |
| Depending on the size of the task use //TODO: comments or open a ticket. | So then you can remind yourself and others about a small task (like refactoring a function, or updating a comment). For larger tasks use //TODO(#3456) which is enforced by a lint rule and the number is an open ticket. |
| Always comment and keep them relevant as code changes. Remove commented blocks of code. | Your code should be as readable as possible, you should get rid of anything distracting. If you refactored a function, don't just comment out the old one, remove it. |
| Avoid irrelevant or funny comments, logs or naming. | While your build process may(should) get rid of them, sometimes your source code may get handed over to another company/client and they may not share the same banter. |
| Make your names search-able with meaningful distinctions avoid shortened names. For functions Use long, descriptive names. A function name should be a verb or a verb phrase, and it needs to communicate its intention. | It makes it more natural to read the source code. |
| Organize your functions in a file according to the step-down rule. Higher level functions should be on top and lower levels below. | It makes it more natural to read the source code. |

**8. Logging**

|  |  |
| --- | --- |
| **What** | **Why** |
| Avoid client-side console logs in production | Even though your build process can(should) get rid of them, but make sure your code style check gives your warning about console logs. |
| Produce readable production logging. Ideally use logging libraries to be used in production mode (such as [winston](https://github.com/winstonjs/winston) or [node-bunyan](https://github.com/trentm/node-bunyan)). | It makes your troubleshooting less unpleasant with colorization, timestamps, log to a file in addition to the console or even logging to a file that rotates daily. [Read more...](https://blog.risingstack.com/node-js-logging-tutorial/) |

**9. API**

**9.1 API design**

Why: Because we try to enforce development of sanely constructed RESTful interfaces, which team members and clients can consume simply and consistently.

Why: Lack of consistency and simplicity can massively increase integration and maintenance costs. Which is why API design is included in this document.

* We mostly follow resource-oriented design. It has three main factors: resources, collection, and URLs
  + A resource has data, gets nested, and there are methods that operate against it
  + A group of resources is called a collection.
  + URL identifies the online location of resource or collection.

Why: This is a very well-known design to developers (your main API consumers). The core idea of REST is the resource and each resource is identified by a URL, and you retrieve that resource by sending a GET request to that URL. Very simple.

* Always use a plural nouns for naming a url pointing to a collection: /users .
* Always use a singular concept that starts with a collection and ends to an identifier:
  + /students/245743
  + /airports/kjfk
* Keep verbs out of your resource URLs.

Why: because we use verbs for something else

* Use verbs for non-resources. In this case, your API doesn't return any resources. Instead, you execute an operation and return the result. These are not CRUD (create, retrieve, update, and delete) operations:
  + GET `/translate?text=Hallo`

Why: Because for CRUD we use HTTP methods on resource or collection URLs. The verbs we were talking about are actually Controllers . You usually don't develop many of these. read more...

* The request body or response type is JSON then please follow camelCase for JSON property names to maintain the consistency.

Why: This is a JavaScript project guideline, Where Programming language for generating JSON as well as Programming language for parsing JSON are assumed to be JavaScript.

* Even though a resource is a singular concept that is similar to an object instance or database record, you should not use your table\_name for a resource name and column\_name resource property.

Why: Because your intention is to expose Resources, not your database schema details

* Again, only use nouns in your URL when naming your resources and don’t try to explain their functionality.

Why: Only use nouns in your resource URLs, avoid endpoints like /addNewUser or /updateUser . Also avoid sending resource operations as a parameter.

* Explain the CRUD functionalities using HTTP methods:

How:

* GET : To retrieve a representation of a resource.
* POST : To create new resources and sub-resources
* PUT : To update existing resources
* PATCH : To update existing resources. It only updates the fields that were supplied, leaving the others alone
* DELETE : To delete existing resources
* For nested resources, use the relation between them in the URL. For instance, using id to relate an employee to a company.

Why: This is a natural way to make resources explorable.

How:

* GET /schools/2/students , should get the list of all students from school 2
* GET /schools/2/students/31 , should get the details of student 31, which belongs to school 2
* DELETE /schools/2/students/31 , should delete student 31, which belongs to school 2
* PUT /schools/2/students/31 , should update info of student 31, Use PUT on resource-URL only, not collection
* POST /schools , should create a new school and return the details of the new school created. Use POST on collection-URLs
* Use a simple ordinal number for version with a v prefix (v1, v2). Move it all the way to the left in the URL so that it has the highest scope:
  + http://api.domain.com/v1/schools/3/students

Why: When your APIs are public for other third parties, upgrading the APIs with some breaking change would also lead to breaking the existing products or services using your APIs. Using versions in your URL can prevent that from happening. read more...

* Response messages must be self descriptive. A good error message response might look something like this:

{

"code": 1234,

"message" : "Something bad happened",

"description" : "More details"

}

or for validation errors:

{

"code" : 2314,

"message" : "Validation Failed",

"errors" : [

{

"code" : 1233,

"field" : "email",

"message" : "Invalid email"

},

{

"code" : 1234,

"field" : "password",

"message" : "No password provided"

}

]

}

Why: Developers depend on well-designed errors at the critical times when they are troubleshooting and resolving issues after the applications they've built using your APIs are in the hands of their users.

Note: Keep security exception messages as generic as possible. For instance, Instead of saying ‘incorrect password’, you can reply back saying ‘invalid username or password’ so that we don’t unknowingly inform user that username was indeed correct and only password was incorrect.

* Use only these 8 status codes to send with you response to describe whether everything worked, The client app did something wrong or The API did something wrong.
  + 200 OK response represents success for GET , PUT or POST requests.
  + 201 Created for when new instance is created. Creating a new instance, using POST method returns 201 status code.
  + 304 Not Modified response is to minimize information transfer when the recipient already has cached representations
  + 400 Bad Request for when the request was not processed, as the server could not understand what the client is asking for
  + 401 Unauthorized for when the request lacks valid credentials and it should re-request with the required credentials.
  + 403 Forbidden means the server understood the request but refuses to authorize it.
  + 404 Not Found indicates that the requested resource was not found.
  + 500 Internal Server Error indicates that the request is valid, but the server could not fulfil it due to some unexpected condition.

Why: Most API providers use a small subset HTTP status codes. For example, the Google GData API uses only 10 status codes, Netflix uses 9, and Digg, only 8. There are over 70 HTTP status codes. However, most developers don't have all 70 memorized. So if you choose status codes that are not very common you will force application developers away from building their apps and over to wikipedia to figure out what you're trying to tell them. read more...

* Provide total numbers of resources in your response
* Accept limit and offset parameters
* The amount of data the resource exposes should also be taken into account. The API consumer doesn't always need the full representation of a resource.Use a fields query parameter that takes a comma separated list of fields to include:
  + GET /student?fields=id,name,age,class
* Pagination, filtering and sorting don’t need to be supported from start for all resources. Document those resources that offer filtering and sorting.

**9.2 API security**

**9.2.1 TLS**

To secure your web API authentication, all authentications should use SSL. OAuth2 requires the authorization server and access token credentials to use TLS. Switching between HTTP and HTTPS introduces security weaknesses and best practice is to use TLS by default for all communication. Throw an error for non-secure access to API URLs.

**9.2.2 Rate limiting**

If your API is public, you may want to consider Rate Limiting.

Why; To protect your APIs from bot threats that call your API thousands of times per hour. You should consider implementing rate limit early on.

**9.2.3 Input Validation**

It's difficult to perform most attacks if the allowed values are limited.

* Validate required fields, field types (e.g. string, integer, boolean, etc), and format requirements. Return 400 Bad Request with details about any errors from bad or missing data.
* Escape parameters that will become part of the SQL statement to protect from SQL injection attacks
* As also mentioned before, don't expose your database scheme when naming your resources and defining your responses

9.2.4 URL validations

* Attackers can tamper with any part of an HTTP request, including the URL, query string,

**9.2.5 Validate incoming content-types.**

The server should never assume the Content-Type. A lack of Content-Type header or an unexpected Content-Type header should result in the server rejecting the content with a 406 Not Acceptable response.

**9.2.6 JSON encoding**

A key concern with JSON encoders is preventing arbitrary JavaScript remote code execution within the browser or node.js, on the server. Use a JSON serialiser to entered data to prevent the execution of user input on the browser/server.

**9.3 API documentation**

* Fill the API Reference section in README.md template for API.
* Describe API authentication methods with a code sample
* Explaining The URL Structure (path only, no root URL) including The request type (Method)

For each endpoint explain:

* URL Params If URL Params exist, specify them in accordance with name mentioned in URL section

Required: id=[integer]

Optional: photo\_id=[alphanumeric]

* If the request type is POST, provide a working examples. URL Params rules apply here too. Separate the section into Optional and Required.
* Success Response, What should be the status code and is there any return data? This is useful when people need to know what their callbacks should expect! Code: 200

Content: { id : 12 }

* Error Response, Most endpoints have many ways to fail. From unauthorised access, to wrongful parameters etc. All of those should be listed here. It might seem repetitive, but it helps prevent assumptions from being made. For example

{

"code": 403,

"message" : "Authentication failed",

"description" : "Invalid username or password"

}

**9.3.1 API design tools**

There are lots of open source tools for good documentation such as API Blueprint and Swagger.