# **Coding Standards**

| **Standard Name** | **Description** |
| --- | --- |
| Project Structure | Standard layout and naming for projects, most likely to be .Net Core for backends and Node JS/Webpack for UI |
| Naming conventions | Consistent naming conventions across all code, domain and entity naming should also match names used by the business |
| Database standards | Sql and NoSql database standards |
| Unit test coverage | Needs to be 80% plus |
| TDD | Test driven design on all new features |
| Validation | Consistent validation of everything that enters any service, with enough information for the consumer to correct any validation errors. |
| API Standards | Highly RESTful, consistent use of verbs and status codes. self-documenting |
| HTTP Status Code |  |

# 

# 

# **Coding Standards(Node Js)**

**Reference :** [**https://github.com/felixge/node-style-guide**](https://github.com/felixge/node-style-guide)

1. **Formatting**

* 4 spaces for indentation
* No trailing white space
* Use semicolons
* 80 characters per line

1. **Use single quotes**

Use single quotes, unless you are writing JSON. This helps you separate your objects’ strings from normal strings.

1. **Opening braces go on the same line**

Your opening braces go on the same line as the statement.

if (true) {

console.log('winning');

}

1. **Use the === operator**
2. **Use Async Await, avoid callbacks**

using async-await which provides a much more compact and familiar code syntax like try-catch.

1. **Use arrow function expressions (=>)**
2. **Declare one variable per let / const statement**
3. **Naming Conventions**

Use lowerCamelCase for variables, properties and function names

Use UpperCamelCase for class names and interface names

Use UPPERCASE for Constants

1. **Design Pattern**

Whenever ever use a design pattern, always suffix class with pattern names like TokenBuilder, UserRepository, ConnectionFactory, LoggerAdapter, TestCaseBuilder etc.

1. **Dependency Injection** - use interface injection on required classes for unit testable.
2. **Unit Testing** - All code should be tested unit tested with unit tests in Given, When, Then format Or AAA (Arrange, Act, Assert)

Use testing frameworks like [Mocha](https://mochajs.org/), Sinon & [Chai](https://www.chaijs.com/)

1. **Error Handling Practices**
2. Use Async-Await or promises for async error handling

Handling async errors in callback style is probably the fastest way to the pyramid of doom. Use a reputable promise library or async-await instead

which enables a much more compact and familiar code syntax like try-catch.

1. Use only the built-in Error object

Many throw errors as a string or as some custom type – this complicates the error handling logic and the interoperability between modules. Whether you reject a promise, throw an exception or emit an error

using only the built-in Error object will increase uniformity and prevent loss of information.

**13. Validating your Typescript**

Use Prettier and TSLint to enforce coding standards on Typescript.Prettier will automatically format your code to conform to style standards.TSLint will perform static analysis to check code for readability, maintainability, and functionality error

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# **Coding Conventions C#**

Reference : <https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions>

**Naming conventions**

There are several naming conventions to consider when writing C# code.

One of the best guides for how to name code is “Clean Code” by Robert C Martin

For general advice on naming this article is quite good

<https://medium.com/coding-skills/clean-code-101-meaningful-names-and-functions-bf450456d90c>

**Pascal case**

You should use pascal casing ("PascalCasing") when naming a class, record, or struct.

**public** **class** DataService

{

}

public record PhysicalAddress(

string Street,

string City,

string StateOrProvince,

string ZipCode);

**public** **struct** ValueCoordinate

{

}

When naming an interface, use pascal casing in addition to prefixing the name with an I. This clearly indicates to consumers that it's an interface.

public interface IWorkerQueue

{

}

When naming public members of types, such as; fields, properties, events, methods, and local functions use pascal casing.

public class ExampleEvents

{

// A public field, these should be used sparingly

public bool IsValid;

// An init-only property

public IWorkerQueue WorkerQueue { get; init; }

// An event

public event Action EventProcessing;

// Method

public void StartEventProcessing()

{

// Local function

static int CountQueueItems() => WorkerQueue.Count;

// ...

}

}

When writing positional-records, use pascal casing for parameters as they're the public properties of the record.

public record PhysicalAddress(

string Street,

string City,

string StateOrProvince,

string ZipCode);

For more information on positional-records, see [Positional syntax for property definition](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record#positional-syntax-for-property-definition).

**Camel case**

You should use camel casing ("camelCasing") when naming private or internal fields, and they should be prefixed with \_.

public class DataService

{

private IWorkerQueue \_workerQueue;

}

Tip

When editing C# code that follows these naming conventions in an IDE that supports statement completion, typing \_ will show all of the object-scoped members.

When working with static fields that are private or internal, use the s\_ prefix and for thread static use t\_.

public class DataService

{

private static IWorkerQueue s\_workerQueue;

[ThreadStatic]

private static TimeSpan t\_timeSpan;

}

When writing method parameters, use camel casing.

public T SomeMethod<T>(int someNumber, bool isValid)

{

}

For more information on C# naming conventions, see [C# Coding Style](https://github.com/dotnet/runtime/blob/main/docs/coding-guidelines/coding-style.md).

**Additional naming conventions**

* Examples that don't include [using directives](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-directive), use namespace qualifications. If you know that a namespace is imported by default in a project, you don't have to fully qualify the names from that namespace. Qualified names can be broken after a dot (.) if they are too long for a single line, as shown in the following example.
* var currentPerformanceCounterCategory = new System.Diagnostics.
* PerformanceCounterCategory();
* You don't have to change the names of objects that were created by using the Visual Studio designer tools to make them fit other guidelines.

**Layout conventions**

Good layout uses formatting to emphasise the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

* Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see [Options, Text Editor, C#, Formatting](https://docs.microsoft.com/en-us/visualstudio/ide/reference/options-text-editor-csharp-formatting).
* Write only one statement per line.
* Write only one declaration per line.
* If continuation lines are not indented automatically, indent them one tab stop (four spaces).
* Add at least one blank line between method definitions and property definitions.
* Use parentheses to make clauses in an expression apparent, as shown in the following code.

if ((val1 > val2) && (val1 > val3))

{

// Take appropriate action.

}

**Commenting conventions**

* Place the comment on a separate line, not at the end of a line of code.
* Begin comment text with an uppercase letter.
* End comment text with a period.
* Insert one space between the comment delimiter (//) and the comment text, as shown in the following example.

// The following declaration creates a query. It does not run

// the query.

* Don't create formatted blocks of asterisks around comments.

Language guidelines

The following sections describe practices that the C# team follows to prepare code examples and samples.

**String data type**

* Use [string interpolation](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated) to concatenate short strings, as shown in the following code.

string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";

* To append strings in loops, especially when you're working with large amounts of text, use a [StringBuilder](https://docs.microsoft.com/en-us/dotnet/api/system.text.stringbuilder) object.

var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";

var manyPhrases = new StringBuilder();

for (var i = 0; i < 10000; i++)

{

manyPhrases.Append(phrase);

}

//Console.WriteLine("tra" + manyPhrases);

**Implicitly typed local variables**

* Use [implicit typing](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables) for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.

var var1 = "This is clearly a string.";

var var2 = 27;

* Don't use [var](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/var) when the type is not apparent from the right side of the assignment. Don't assume the type is clear from a method name. A variable type is considered clear if it's a new operator or an explicit cast.

int var3 = Convert.ToInt32(Console.ReadLine());

int var4 = ExampleClass.ResultSoFar();

* Don't rely on the variable name to specify the type of the variable. It might not be correct. In the following example, the variable name inputInt is misleading. It's a string.

var inputInt = Console.ReadLine();

Console.WriteLine(inputInt);

* Avoid the use of var in place of [dynamic](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types). Use dynamic when you want run-time type inference. For more information, see [Using type dynamic (C# Programming Guide)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/using-type-dynamic).
* Use implicit typing to determine the type of the loop variable in [for](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/for) loops.

The following example uses implicit typing in a for statement.

var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";

var manyPhrases = new StringBuilder();

for (var i = 0; i < 10000; i++)

{

manyPhrases.Append(phrase);

}

//Console.WriteLine("tra" + manyPhrases);

* Don't use implicit typing to determine the type of the loop variable in [foreach](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/foreach-in) loops.

The following example uses explicit typing in a foreach statement.

foreach (char ch in laugh)

{

if (ch == 'h')

Console.Write("H");

else

Console.Write(ch);

}

Console.WriteLine();

Note

Be careful not to accidentally change a type of an element of the iterable collection. For example, it is easy to switch from [System.Linq.IQueryable](https://docs.microsoft.com/en-us/dotnet/api/system.linq.iqueryable) to [System.Collections.IEnumerable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable) in a foreach statement, which changes the execution of a query.

**Unsigned data types**

In general, use int rather than unsigned types. The use of int is common throughout C#, and it is easier to interact with other libraries when you use int.

**Arrays**

Use the concise syntax when you initialise arrays on the declaration line. In the following example, note that you can't use var instead of string[].

string[] vowels1 = { "a", "e", "i", "o", "u" };

If you use explicit instantiation, you can use var.

var vowels2 = new string[] { "a", "e", "i", "o", "u" };

If you specify an array size, you have to initialise the elements one at a time.

var vowels3 = new string[5];

vowels3[0] = "a";

vowels3[1] = "e";

// And so on.

**Delegates**

Use Func<> and Action<> instead of defining delegate types. In a class, define the delegate method.

public static Action<string> ActionExample1 = x => Console.WriteLine($"x is: {x}");

public static Action<string, string> ActionExample2 = (x, y) =>

Console.WriteLine($"x is: {x}, y is {y}");

public static Func<string, int> FuncExample1 = x => Convert.ToInt32(x);

public static Func<int, int, int> FuncExample2 = (x, y) => x + y;

Call the method using the signature defined by the Func<> or Action<> delegate.

ActionExample1("string for x");

ActionExample2("string for x", "string for y");

Console.WriteLine($"The value is {FuncExample1("1")}");

Console.WriteLine($"The sum is {FuncExample2(1, 2)}");

If you create instances of a delegate type, use the concise syntax. In a class, define the delegate type and a method that has a matching signature.

public delegate void Del(string message);

public static void DelMethod(string str)

{

Console.WriteLine("DelMethod argument: {0}", str);

}

Create an instance of the delegate type and call it. The following declaration shows the condensed syntax.

Del exampleDel2 = DelMethod;

exampleDel2("Hey");

The following declaration uses the full syntax.

Del exampleDel1 = new Del(DelMethod);

exampleDel1("Hey");

try**-**catch **and** using **statements in exception handling**

* Use a [try-catch](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch) statement for most exception handling.

static string GetValueFromArray(string[] array, int index)

{

try

{

return array[index];

}

catch (System.IndexOutOfRangeException ex)

{

Console.WriteLine("Index is out of range: {0}", index);

throw;

}

}

* Simplify your code by using the C# [using statement](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement). If you have a [try-finally](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-finally) statement in which the only code in the finally block is a call to the [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) method, use a using statement instead.

In the following example, the try-finally statement only calls Dispose in the finally block.

Font font1 = new Font("Arial", 10.0f);

try

{

byte charset = font1.GdiCharSet;

}

finally

{

if (font1 != null)

{

((IDisposable)font1).Dispose();

}

}

You can do the same thing with a using statement.

using (Font font2 = new Font("Arial", 10.0f))

{

byte charset2 = font2.GdiCharSet;

}

In C# 8 and later versions, use the new using syntax that doesn't require braces:

using Font font3 = new Font("Arial", 10.0f);

byte charset3 = font3.GdiCharSet;

&& **and** || **operators**

To avoid exceptions and increase performance by skipping unnecessary comparisons, use && instead of & and || instead of | when you perform comparisons, as shown in the following example.

Console.Write("Enter a dividend: ");

int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");

int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor > 0))

{

Console.WriteLine("Quotient: {0}", dividend / divisor);

}

else

{

Console.WriteLine("Attempted division by 0 ends up here.");

}

If the divisor is 0, the second clause in the if statement would cause a run-time error. But the && operator short-circuits when the first expression is false. That is, it doesn't evaluate the second expression. The & operator would evaluate both, resulting in a run-time error when divisor is 0.

new **operator**

* Use one of the concise forms of object instantiation, as shown in the following declarations. The second example shows syntax that is available starting in C# 9.

var instance1 = new ExampleClass();

ExampleClass instance2 = new();

The preceding declarations are equivalent to the following declaration.

ExampleClass instance2 = new ExampleClass();

* Use object initializers to simplify object creation, as shown in the following example.

var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,

Location = "Redmond", Age = 2.3 };

The following example sets the same properties as the preceding example but doesn't use initializers.

var instance4 = new ExampleClass();

instance4.Name = "Desktop";

instance4.ID = 37414;

instance4.Location = "Redmond";

instance4.Age = 2.3;

**Event handling**

If you're defining an event handler that you don't need to remove later, use a lambda expression.

public Form2()

{ this.Click += (s, e) =>

{

MessageBox.Show(

((MouseEventArgs)e).Location.ToString());

};}

The lambda expression shortens the following traditional definition.

public Form1()

{

this.Click += new EventHandler(Form1\_Click);

}

void Form1\_Click(object sender, EventArgs e)

{

MessageBox.Show(((MouseEventArgs)e).Location.ToString());

}

**Static members**

Call [static](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static) members by using the class name: *ClassName.StaticMember*. This practice makes code more readable by making static access clear. Don't qualify a static member defined in a base class with the name of a derived class. While that code compiles, the code readability is misleading, and the code may break in the future if you add a static member with the same name to the derived class.

**LINQ queries**

* Use meaningful names for query variables. The following example uses seattleCustomers for customers who are located in Seattle.

var seattleCustomers = from customer in customers

where customer.City == "Seattle"

select customer.Name;

* Use aliases to make sure that property names of anonymous types are correctly capitalised, using Pascal casing.

var localDistributors =

from customer in customers

join distributor in distributors on customer.City equals distributor.City

select new { Customer = customer, Distributor = distributor };

* Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and a distributor ID, instead of leaving them as Name and ID in the result, rename them to clarify that Name is the name of a customer, and ID is the ID of a distributor.

var localDistributors2 =

from customer in customers

join distributor in distributors on customer.City equals distributor.City

select new { CustomerName = customer.Name, DistributorID = distributor.ID };

* Use implicit typing in the declaration of query variables and range variables.

var seattleCustomers = from customer in customers

where customer.City == "Seattle"

select customer.Name;

* Align query clauses under the [from](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/from-clause) clause, as shown in the previous examples.
* Use [where](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/where-clause) clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

var seattleCustomers2 = from customer in customers

where customer.City == "Seattle"

orderby customer.Name

select customer;

* Use multiple from clauses instead of a [join](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/join-clause) clause to access inner collections. For example, a collection of Student objects might each contain a collection of test scores. When the following query is executed, it returns each score that is over 90, along with the last name of the student who received the score.

var scoreQuery = from student in students

from score in student.Scores

where score > 90

select new { Last = student.LastName, score };

**Best guidelines:**

1. **Classes, Functions, and Methods**

Keep functions, and methods reasonably sized. This depends upon the language being used. A good rule of thumb for module length is to constrain each module to one function or action (i.e. each module should only do one “thing”). The names of the classes, functions, and methods shall have verbs in them. That is the names shall specify an action, e.g. “getName”, “computeTemperature”. As we are using camelCase for naming, we should be follow this throughout the code.

2. **Indentation**

Indentation should be used to:

• Emphasise the body of a control statement such as a loop or a select statement

• Emphasise the body of a conditional statement

• Emphasise a new scope block

A minimum of 4 spaces shall be used to indent. It is important that this indentation amount be consistently applied throughout the program. Tabs shall not be used for indentation purposes.

3. **Inline Comments**

Inline comments explaining the functioning of the subroutine or key aspects of the algorithm shall be frequently used.

4. **Variable Names**

Variables shall have mnemonic or meaningful names that convey to a casual observer, the intent of its use. Variables shall be initialised prior to its first use.

5. **Use of Braces**

In some languages, braces are used to delimit the bodies of conditional statements, control constructs, and blocks of scope. We shall use the following

bracing style:

for (condition)

{

/\* Some work is done here. \*/

}

Braces shall be used even when there is only one statement in the control block.

6. **Spacing**

The proper use of spaces within a line of code can enhance readability. Some rules are as follows:

• A keyword followed by a parenthesis should be separated by a space.

• A blank space should appear after each comma in an argument list.

• All binary operators except “.” should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement(“—“) from their operands.

• Casts should be made followed by a blank space.

7. **Line Length**

It is considered good practice to keep the lengths of source code lines at or below 80 characters. Lines longer than this may not be displayed properly on some terminals and tools.

8. **Use of Parentheses**

It is better to use parentheses liberally. Even in cases where operator precedence unambiguously dictates the order of evaluation of an expression, often it is beneficial from a readability point of view to include parentheses anyway.

9. **Inline Comments**

Inline comments promote program readability. As the name suggests, inline comments appear in the body of the source code itself. They explain the logic or parts of the algorithm which are not readily apparent from the code itself. Inline comments can also be used to describe the task being performed by a block of code.

10. **Version**

Follow the same version of languages/packages/libraries throughout the code. Update the code with the latest version used.

# **TDD (Test Driven Development)**

# 

AAA

AAA stands for Arrange, Act and Assert and is an easy way to think about the arrangement of a unit test.

**Arrange**

This is where the **system under test** (or SUT) is set up ready to be tested. Each suite of unit tests should test one component and one component only. In the example below the SUT is the Calculator and for clarity it is assigned to the variable sut so it is clear exactly what is being tested.

**Act**

This is where the actual system under test is called. This should simulate how it will be used in production code, but it is often useful to pass in potentially exception causing parameters such as null to avoid surprises in production.

**Assert**

This is an essential part of any test and is where the test is declared as either passing or failing

[Fact]

**public** **void** AddTwoNumbers()

{

// Arrange

**var** number1 = 3.0;

**var** number2 = 4.0;

**var** sut = **new** Calculator();

//Act

**var** result = sut.Add(number1, number2);

*//Assert*

Assert.Equal(7.0, result);

}

**public** **class** Calculator

{

**public** **double** Add(**double** number1, **double** number2)

{

**return** number1 + number2;

}

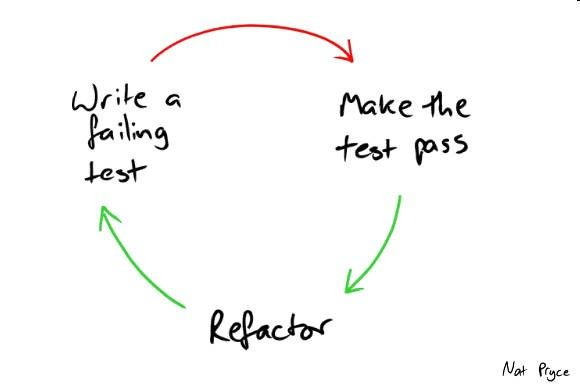
}

**How to approach writing code with TDD**

Often difficult to know where to begin when writing code for a feature, but TDD can help with this. It’s important to always start with a failing test, so think about the simplest part of the feature that you will need and first think about how you would test it.

**RED - GREEN - REFACTOR**

A standard practice with TDD is to use Red, Green, Refactor.



1. Write a failing test that describes the next simplest thing you need your system code to do. Run the test to prove that it fails proving that your system does not yet behave in the way set out in the test.
2. Change the implementation code in the quickest and easiest way in order to make the test pass, but only do the minimum you can to make the test pass.
3. Review and refactor the code you have so far. Remember refactoring simply means improving code quality without changing what it does.

Never write implementation code unless it is to make a failing test pass.

**TOP DOWN APPROACH**

For this we will use an example user story where we need to get all roles for a tenant.

Start by mapping out the process at a very high level, this will form the basis of a class that will be the “orchestrator”. With the single responsibility principle no class should have more than one responsibility, and an “orchestrator” just organises the work, nothing else.

**Unit Testing Best Practices**

* Unit Test cases should be independent. In case of any enhancements or change in requirements, unit test cases should not be affected.
* Test only one thing at a time.
* Follow clear and consistent naming conventions for your unit tests
* In case of a change in code in any module, ensure there is a corresponding unit Test Case for the module, and the module passes the tests before changing the implementation
* Bugs identified during unit testing must be fixed before proceeding to the next phase in SDLC
* Adopt a "test as your code" approach. The more code you write without testing, the more paths you have to check for errors.

# **API Standards**

**Use Nouns instead of Verbs**

In API development the REST approach can be called resource based. So in your application, you work with resources and their collections (eg. a single book, and a list of all books). The actions on the resources are strictly defined by the HTTP methods such as GET, PUT, POST, PATCH, DELETE (and a few others, that are less relevant for this example) and only they should be used to change the state of the resource. This leads us to the endpoint URI construction. Taking the above into account properly constructed endpoint should look similar to this:

Correct way

GET /books/123

DELETE /books/123

POST /books

PUT /books/123

PATCH /books/123

-vs-

Incorrect way

GET /addBook123 (by the way, GET should be only used to READ data and never to change its state in any way)

GET /DeleteBooks/123

POST /DeleteAllBooks

POST /books/123/delete

Name the collections using Plural Nouns

Eg

Get /tenants/{id}

not

Get tenant/{id}

Use resource nesting to show relations or hierarchy

Resource objects often have some kind of functional hierarchy or are related to each other. For example in the online store, we have ‘users’ and ‘orders’. Orders always belong to some user, therefore we may have the following endpoints structure laid out:

/users <- user’s list

/users/123 <- specific user

/users/123/orders <- orders list that belongs to a specific user

/users/123/orders/0001 <- specific order of a specific user

**Query / Command Segregation**

In software design there is the concept of a query and a command.

A query returns an object/list but does not have any side effects (does not alter the state of the system in any way)

A command cause a side effect (changed the system) but does not return anything - it returns void

Http Verbs are designed with this principal in mind.

GET - These are queries, they are used to get an object/list but should never be used to change the state of the system.

POST, PUT, DELETE - These are commands, they change the state of the system but should not return an object/list

If an object is created using a POST the response should return a “Location” header with the url of the newly created entity

**Error Handling**

Errors can be split into 2 main categories

**Client Errors - the caller has done something**

* Not passed a valid auth token (401)
* Has a request a resource they are not allowed access too (403)
* Has passed a payload that has failed validation (400)

Client errors can be resolved by the client and the API should return enough information that it is clear what steps can be taken to resolve the error. For instance if validation fails the API should explain what field failed and why and what should be changed to resolve the issue.

**Server Errors - the caller is not at fault, the server is having problems**

* Service or a dependency such as a database is not available (503)

These errors are usually temporary in nature but here is nothing the client can do other than retry later.

**Filtering, sorting, paging, and field selection**

Few of the most important features for consuming an API are filtering, sorting and paging. Resource collections are oftentimes enormous, and when some data has to be retrieved from them, it would be simply not very efficient to always get the full list and browse it for specific items. Therefore we can use:

* **Filtering** - to narrow down the query results by specific parameters, eg. creation date, or country
* **Sorting** - basically allows sorting the results ascending or descending by a chosen parameter or parameters, eg. by date
* **Paging** - uses “limit” in order to narrow down the number of results shown to a specific number, and “offset” to specify which part of the results range to be shown - this is important in cases where the number of total results is greater than the one presented, this works like pagination you may encounter on many websites

Filtering, sorting, and paging are used by adding a so-called query parameter to the endpoint that is being called. These may look as follows:

**Filtering:**

GET /users?country=USA

GET /users?creation\_date=2019-11-11

GET /users?creation\_date=2019-11-11

Sorting:

GET /users?sort=birthdate\_date:asc

GET /users?sort=birthdate\_date:desc

Paging:

GET /users?limit=100

GET /users?offset=2

All together:

GET /users?country=USA&creation\_date=2019-11-11&sort=birthdate\_date:desc&limit=100&offset=2

All query parameters should be optional and shouldn’t be used to get a single resource

ie never

GET /users?id=1

**API Documentation**

All APIs should be documented using OpenAPI 3.0

This will allow us to view the API using Swagger, eg<https://petstore.swagger.io/>

**Http Status Codes**

**Success Codes**

The request has been fulfilled successfully

| **Code** | **Definition** | **Example** |
| --- | --- | --- |
| **200 - OK** | The request was processed correctly and a payload has been returned. Ideally 200 responses should only be used if there have been no side effects (changes of state) to the underlying system. | Standard for a GET request when data is returned to the caller but is not changed in any way on the underlying data store. |
| **201 - Created** | The request has been fulfilled, resulting in the creation of a new resource. | This would typically be used on a POST if it was creating a new record. |
| **202 - Accepted** | Accepted response status code indicates that the request has been accepted for processing, but the processing has not been completed; in fact, processing may not have started yet | When a request is going to be processed in the background and the outcome is not known when the response is sent back. For example, a message being put on a queue as a result of an API call. |
| **204 No Content** | The server successfully processed the request, and is not returning any content. | This should be returned when a request results in a change of state on the underlying data store, but not a new record. Eg PUT, DELETE or a POST that carries out an action. |

**Client Errors**

The request has not been processed due to reasons with in the calling clients control.

| **400 Bad Request** | The server cannot or will not process the request due to an apparent client error (e.g., malformed request syntax, size too large, invalid request message framing, or deceptive request routing) | The structure or the content of the message fails validation.  The exact reason for the validation failure should be returned in the body in such a way that the caller would be able to resolve the issue. |
| --- | --- | --- |
| **401 Unauthorized** | The user does not have valid authentication credentials for the target resource. | An auth token is either missing or does not pass validation so that the users identity can not be verified. |
| **403 Forbidden** | The request contained valid data and was understood by the server, but the server is refusing action. | The auth token is validated and the server is able to identify the user however the user does not have permission to access this resource.  Note: It may be better to return 404 for security reasons, ie not revealing that a resource exists. |
| **404 Not Found** | The requested resource could not be found but may be available in the future. Subsequent requests by the client are permissible. | Either the api endpoint does not exist or the specific resource being requested doesn’t exist.  If a resource doesn’t exist this should be highlighted in the body. |

**Server Errors**

The request has not been processed due a problem with the server which is outside of the calling clients control.

| **500 Internal Server Error** | A generic error message, given when an unexpected condition was encountered and no more specific message is suitable. | An API endpoint should never return this code. If it should signify that the code is not handling errors correctly. |
| --- | --- | --- |
| **501 Not Implemented** | The server either does not recognize the request method, or it lacks the ability to fulfil the request. | Potentially could be used on stubbed services before the implementation has been completed. |
| **503 Service Unavailable** | The server cannot handle the request (because it is overloaded or down for maintenance). Generally, this is a temporary state. | If a dependency is unavailable such as a database then this code should be returned with further, but very limited, information in the body. |