Guidelines for code comments in grafana-* packages

This document aims to give you some recommendation on how to add code comments to the exported code in the grafana packages.

Table of Contents

Add package description

Each package has an overview explaining the overall responsibility and usage of the package.

You can document this description with `@packageDocumentation` tag.

Add this tag to the `<packageRoot>/src/index.ts` entry file to have one place for the package description.

Set stability of an API

All `exported` apis from the package should have a release tag to indicate its stability.

- `@alpha` - early draft of api and will probably change.
- `@beta` - close to being stable but might change.
- `@public` - ready for usage in production.
- `@internal` - for internal use only.

Main stability of APIs

Add a tag to mark the stability of the whole exported `class/interface/function/type` etc.

Please place the `release tag` at the bottom of the comment to make it consistent among files and easier to read.

Do:

```
/**
 * Will help to create DataFrame objects and handle
 * the heavy lifting of creating a complex object.
```

```
 *
 * @example
 * ```typescript
 * const dataFrame = factory.create();
 * ```
 *
 * @public
 **/
export class DataFrameFactory {
  create(): DataFrame {}
}
```

Don't

```
/**
 * Will help to create DataFrame objects and handle
 * the heavy lifting of creating a complex object.
 *
 * @public
 * @example
 * ```typescript
 * const dataFrame = factory.create();
 * ```
 **/
export class DataFrameFactory {
  create(): DataFrame {}
}
```

Partial stability of APIs

Add the main stability of the API at the top according to Main stability of API.

Then override the non-stable parts of the API with the proper release tag. This should also be place at the bottom of the comment block.

Do:

```
/**
 * Will help to create DataFrame objects and handle
 * the heavy lifting of creating a complex object.
 *
 * @example
 * ```typescript
 * const dataFrame = factory.create();
 * ```
 *
 * @public
 **/
```

```
export class DataFrameFactory {
  create(): DataFrame {}

  /**
   * @beta
   **/
  createMany(): DataFrames[] {}
}
```

Don't

```
/**
 * Will help to create DataFrame objects and handle
 * the heavy lifting of creating a complex object.
 *
 * @example
 * ```typescript
 * const dataFrame = factory.create();
 * ```
 **/
export class DataFrameFactory {
  /**
   * @public
   **/
  create(): DataFrame {}

  /**
   * @beta
   **/
  createMany(): DataFrame[] {}
}
```

Deprecate an API

If you want to mark an API as deprecated to signal that this API will be removed in the future, then add the `@deprecated` tag.

If applicable add a reason why the API is deprecated directly after the `@deprecated tag`.

Specify parameters

If you want to specify the possible parameters that can be passed to an API, then add the `@param` tag.

This attribute can be skipped if the type provided by `typescript` and the function comment or the function name is enough to explain what the parameters are.

Do:

```
/**
 * Will help to create a resource resolver depending
 * on the current execution context.
 *
 * @param context - The current execution context.
 * @returns FileResolver if executed on the server otherwise a HttpResolver.
 * @public
 **/
export const factory = (context: Context): IResolver => {
  if (context.isServer) {
    return new FileResolver();
  }
  return new HttpResolver();
};
```

Don't

```
/**
 * Will compare two numbers to see if they are equal to each others.
 *
 * @param x - The first number
 * @param y - The second number
 * @public
 **/
export const isEqual = (x: number, y: number): boolean => {
  return x === y;
};
```

Set return values

If you want to specify the return value from a function you can use the `@returns` tag.

This attribute can be skipped if the type provided by `typescript` and the function comment or the function name is enough to explain what the function returns.

Do:

```
/**
 * Will help to create a resource resolver depending
 * on the current execution context.
 *
 * @param context - The current execution context.
 * @returns FileResolver if executed on the server otherwise a HttpResolver.
 * @public
 **/
```

```ts
export const factory = (context: Context): IResolver => {
  if (context.isServer) {
    return new FileResolver();
  }
  return new HttpResolver();
};
```

Don't

```ts
/**
 * Will compare two numbers to see if they are equal to each others.
 *
 * @returns true if values are equal
 * @public
 **/
export const isEqual = (x: number, y: number): boolean => {
  return x === y;
};
```