



# Introduction

In this **project** you will develop a CLI program developed with node.js that you will be able to run using a terminal application. The program will have to connect to a third-party API to show the data returned and store it in the local system.

## What are the main objectives in this project?

- Learn how to connect to a third party API using node.js
- Learn how to implement a CLI program using node.js
- Learn how to work with the filesystem apis of node.js
- Learn how to interact with a CLI program and develop a menu that users can use to know how to execute the program

# 1. General analysis

## 1.1 Phase 1

In this phase you will have to implement the options of the CLI program so that you can perform network requests.

### 1.1.1 API key

Before you can begin making requests to the API (<https://api.themoviedb.org/3>) you will have to read the documentation so that you can learn how to get an API Key.

To learn more about the API visit the following link:

<https://www.themoviedb.org/documentation/api>

Once you have created an account and you have an API Key, you can begin working with the API.

**You have to remember that you will need to include the API Key with each request you make to the API.**

### 1.1.2 Storing the api key as an environment variable

Once you have a key, it is important to remember that API Keys should be kept private so that others cannot use them. A best practice to work with API Keys is to store them in environment variables and then in your code you will use these variables instead of the value of the API Key directly.

In order to store the key in an environment variable in node.js you need to follow the following steps:

1. Create an **.env** file (with this name) where you will store the **API\_KEY** entry in the **"src"** folder of your project.

```
API_KEY=eyJhbGciOiJIUzI1NiJ9.eyJhdWQiOiI5YmIyYTI1MmFlNWY3MTkzYjNkMDExM
```

2. The inputs of the **.env** file as **API\_KEY** will be available in **process.env.API\_KEY** —you can have several entries in the **.env** file, as many as you want, and each will be available under **process.env.THE\_KEY\_NAME**.
3. In the **.env** file is where you should save your **API\_KEY** so that it is not included in your git repository for security reasons so that other people do not use your **API\_KEY**
4. Create a **.gitignore** file that excludes the **.env** file so that you don't include it file in the git history of your repository

Once you have created the **.env** file, you will need to use the **dotenv** npm package so that all the entries in the **.env** file are available under **process.env**. For this, you will have to learn how to work with the **dotenv** npm package.

### 1.1.3 The CLI Program

The first step is to create the main executable file of the pill that should be named **moviedb.js**. In this file you will have to implement all the commands of the CLI program and the options of each command using the **commander** package.

<https://github.com/tj/commander.js#readme>

Furthermore, the command will need to have execute permissions so that you can execute it from the terminal like so: `./moviedb.js commands --options`

### 1.1.4 Ensuring that the file can be executed from the command line using the node.js environment

Before executing the command line file, you should research how to be able to execute the main JavaScript file in the following way:

```
./moviedb.js command... ---options... --flags...
```

### 1.1.5 Popular persons

URL: <https://api.themoviedb.org/3/person/popular?page=1>

In this part of the pill you will have to implement the `get-persons` command that you can specify when you execute the `moviedb.js` file. This should be done using the `commander` package and the `.command(...)` method it provides.

The `get-persons` command should have the following required fields:

- **description:** `Make a network request to fetch the most popular persons`
  - the description of the command that will be shown in the help menu so that the users of the CLI program can know what the command is used for at all times
- **options**
  - `--popular`
    - shorthand version: `-p`
    - required: `yes`

- description: "Fetch the popular persons"
- `--page`
  - shorthand version: **none**
  - input type: **number**
  - required: **yes**
  - description: "The page of persons data results to fetch"
  - this option should accept an input value that indicates the page number that should be used as query parameter  
`...popular?page=1` in the URL for pagination

With this command and options in place, you will have to implement the functionality of the `./moviedb` file so that it makes a network request and renders the results in the terminal using several formatting requirements. **The CLI program should be used in the following way:** `./moviedb.js get-persons --page 1 --popular`

#### 1.1.5.1 Starting the terminal spinner

Before making the network request, you will have to use the [ora](#) npm package to run a spinner in the terminal that renders the following message until the request has finished: `Fetching the popular person's data...`

#### 1.1.5.2 Making the network request

The next step is to make a network request using the built in `https.request()` module from node.js. The request should be made to the following url passing in the page as a query param that you will have to use for pagination.

<https://api.themoviedb.org/3/person/popular?page=1>

The URL should be created dynamically depending on the commands and options that the user executing the program has passed when running the application.

If the user has chosen the `get-persons` command with the `--popular` option and the `--page 2` flag, you will have to make a network request to the URL as such:

```
./moviedb.js get-persons --page 2 --popular
```

<https://api.themoviedb.org/3/person/popular?page=2>

In order to learn how to use the request module, you will have to read the following link: [https://nodejs.org/api/https.html#https\\_https\\_request\\_options\\_callback](https://nodejs.org/api/https.html#https_https_request_options_callback)

NOTE:

**You must use the `https.request()` module to make all the network requests, you cannot use other libraries such as `node-fetch` or `axios`.**

### 1.1.5.3 Handling errors when making the request

If the network request fails, you will have to use the `ora.fail()` method to render the error message. This means that you will have to handle and listen for any errors that might be returned from the `https.request()` method and stop the spinner with the error message as the text output.

### 1.1.5.4 Rendering the persons data

Once you have the data from the API, you will have to render the results in the terminal using `console.log` and the [chalk](#) npm package so that you can render text in colors in the terminal.

### 1.1.5.5 Page number for pagination

If the total number of pagination pages is greater than the current page, you will have to render a text output similar to the following using **console.log** and the **chalk.white()** method, otherwise no message should be rendered for pagination.

```
-----  
Page: 1 of: 500
```

### 1.1.5.6 Persons data

Then, for each person entry from the network response, you will have to render using **console.log** and **chalk** the following contents:

- -----
  - a line of dashes to separate the persons from each other
  - using chalk.white
- `\\n`
  - an empty line using an `\\n` character to insert a line break
  - using chalk.white
- Person:
  - using chalk.white and an empty line after it
- ID:
  - using chalk.white to render the id of the person
- Name:
  - using chalk.bold in blue color to render the name of the person
- Department:

- if the `known_for_department` property of the person is `Acting`, you will have to output the text using `chalk.magenta`
- otherwise, no text for the department should be rendered
- Appearing in movies:
  - in this step, you will have to check if the person has any movie with a title that is **not undefined** inside the `known_for` property
  - if the person has any movie with a valid title you will have to render the text title using `chalk.white` and the following:
    - empty line
    - Movie:
      - using `chalk.white`
      - prepended with a tab character ``\t`` to separate the text from the left side of the screen
    - ID:
      - using `chalk.white`
      - prepended with a tab character ``\t``
      - the movie id
    - Release date:
      - using `chalk.white`
      - prepended with a tab character ``\t``
      - the movie's release date
    - Title:
      - using `chalk.white`
      - prepended with a tab character ``\t``
      - the movie title
    - empty line



- if the person doesn't have any movies with a valid title inside the `known_for` property you will have to render the following:
  - person name
  - +
  - doesn't appear in any movie
  - +
  - empty line
  - example: `Alex doesn't appear in any movie\n``

#### 1.1.5.7 Ending the terminal spinner

Once all the data has been rendered to the terminal, you will have to execute the `ora.succeed()` method to stop the spinner with the following text:

✓Popular Persons data loaded

#### 1.1.5.8 Example output of the data

```
Person:

ID: 57755
Name: Woody Harrelson
Department: Acting

Appearing in movies:

    Movie:
    ID: 70160
    Release Date: 2012-03-12
    Title: The Hunger Games
```

-----  
Person:

ID: 30084

Name: Anna Torv

Department: Acting

Anna Torv doesn't appear in any movie

✓ Popular Persons data loaded

## 1.1.6 Person details

URL: <https://api.themoviedb.org/3/person/:id>

In this part of the pill you will have to implement the **get-person command** that you can specify when you execute the **moviedb.js** file. This should be done using the **commander** package and the **.command(...)** method it provides.

This command will be used to load the information of a single person based on the **id** you can get after executing the **get-persons command** in the first step.

The **get-person** command should have the following required fields:

- **description:** **Make a network request to fetch the data of a single person**
  - the description of the command that will be shown in the help menu so that the users of the CLI program can know what the command is used for at all times
- **options**
  - **--id**
    - shorthand version: **-i**
    - required: **yes**
    - description: **"The id of the person"**
    - this option should accept an input value that indicates the **id** of the person that you want to get the details of

With this command and options in place, you will have to implement the functionality of the **./moviedb** file so that it makes a network request and renders the results in the terminal using several formatting requirements.

The CLI program should be used in the following way:

```
./moviedb get-person --id XXXXXXXX
```

#### 1.1.6.1 Starting the terminal spinner

Before making the network request, you will have to use the [ora](#) npm package to run a spinner in the terminal that renders the following message until the request has finished: **Fetching the person data...**

#### 1.1.6.2 Making the network request

The next step is to make a network request to the following url passing in the person id received from the **--id** option of the command:

<https://api.themoviedb.org/3/person/:personId>

The URL should be created dynamically depending on the **--id** option that the user executing the program has passed when running the application.

If the user has chosen the **get-person** command with the **--id** option and the **990393** value, you will have to make a network request to the URL as such:

```
./moviedb.js get-person --id 990393
```

<https://api.themoviedb.org/3/person/990393>

#### 1.1.6.3 Handling errors when making the request

If the network request fails, you will have to use the `ora.fail()` method to render the error message. This means that you will have to handle and listen for any errors that might be returned from the `https.request()` method and stop the spinner with the error message as the text output.

#### 1.1.6.4 Rendering the person data

Once you have the data from the API, you will have to render the results in the terminal using **console.log** and the [chalk](#) npm package so that you can render text in colors in the terminal.

#### 1.1.6.5 Person data

Then, once you have the data from the network response, you will have to render it using **console.log** and **chalk**:

- `\n-----`
  - a line of dashes to separate the movies from each other prepended by an empty line
  - using `chalk.white`
- Person:
  - using `chalk.white` and an ``\n`` character to insert a line break
- ID:
  - using `chalk.white` to render the id of the person
- Name:
  - using `chalk.bold` in blue color to render the name of the person
- Birthday:
  - To create the birthday message you should concatenate the following strings to form the message:
    - using `chalk.white` to render the person's birthday
    - using `chalk.gray` and an ``|`` character
    - using `chalk.white` to render the place of the birth found in the **place\_of\_birth** property

- **Example:**

- **Birthday: 1976-10-23 | Vancouver, British Columbia, Canada**

- **Department:**

- if the **known\_for\_department** property of the person is **Acting**, you will have to output the text using `chalk.magenta`
- otherwise, no text for the department should be rendered

- **Biography:**

- using `chalk.bold` in blue color to render the biography of the person

- **Also known as:**

- in this step, you will have to check if the **also\_known\_as** property of the person is not empty so that you can output the following:

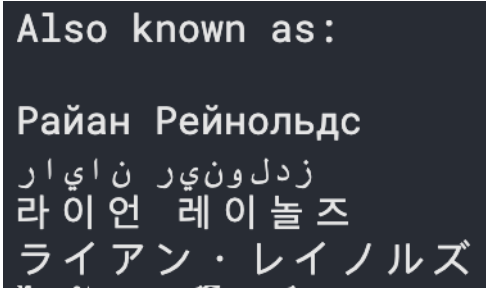
- an ``\n`` character to insert a line break

- Also known as:

- using `chalk.white` and an ``\n`` character to insert a line break

- Then, for each entry in the **also\_known\_as** array, you should render it using `chalk.white` to have a list of all the aliases of the person. Each alias should be followed by a ``\n`` character

- **Example:**

- 

- if the person doesn't have any entries in the **also\_known\_as** array, you will have to render the following:

- an ``\n`` character to insert a line break

- using chalk.yellow to render the name of the person
- +
- `doesn't have any alternate names`
- +
- empty line
- example: `Alex doesn't have any alternate names\n`

#### 1.1.6.6 Ending the terminal spinner

Once all the data has been rendered to the terminal, you will have to execute the `ora.succeed()` method to stop the spinner with the following text:

✓ Person data loaded

#### 1.1.6.7 Example output of the data of a single person

```
| Fetching the person data...
-----
Person:

ID: 990393
Name: Erin Moriarty
Birthday: 1994-06-24 | New York City, USA
Department: Acting
Biography: Erin Moriarty (born June 24, 1994) is an American actress. She has had recurring roles on One Life to Live and True Detective, been a series regular on Red Widow and Jessica Jones and appeared in supporting roles in several movies.

Also known as:

Εριν Μοριάρτι
✓ Person data loaded
```

## 1.2 Phase 2 (Optional)

In this phase of the pill you will have to implement the commands to fetch the movie data from the MovieDB API.

### 1.2.1 Movies

URL: <https://api.themoviedb.org/3/movie/popular?page=1>

In this part of the pill you will have to implement the **get-movies command** that you can specify when you execute the **moviedb.js** file. This should be done using the **commander** package and the **.command(...)** method it provides.

The **get-movies** command should have the following required fields:

- **description:** **Make a network request to fetch movies**
  - the description of the command that will be shown in the help menu so that the users of the CLI program can know what the command is used for at all times
- **options**
  - **--page**
    - shorthand version: **none**
    - input type: **number**
    - required: **yes**
    - description: **"The page of movies data results to fetch"**
    - this option should accept an input value that indicates the page number that should be used as query parameter  
**...popular?page=1** in the URL for pagination



- `--popular`
  - shorthand version: `-p`
  - required: **no**
  - description: “Fetch the popular movies”
  - This option will indicate the command that it should fetch the data from the popular movies URL
  - You will need to read the API docs to see what endpoint you can connect to in order to fetch the popular movies
- `--now-playing`
  - shorthand version: `-n`
  - required: **no**
  - description: “Fetch the movies that are playing now”
  - This option will indicate the command that it should fetch the data from the movies now playing URL
  - You will need to read the API docs to see what endpoint you can connect to in order to fetch the movies playing now

With this command and options in place, you will have to implement the functionality of the `./moviedb` file so that it makes a network request and renders the results in the terminal using several formatting requirements.

**The CLI program should be used in the following way:**

- `./moviedb.js get-movies --popular --page 2`
  - This will fetch the movies data from the popular movies endpoint of the API with a pagination option that indicates which page to fetch the data from
- `./moviedb.js get-movies --now-playing --page 2`

- This will fetch the movies data from the movies that are playing now endpoint of the API with a pagination option that indicates which page to fetch the data from
- If neither the `--popular` or `--now-playing` options are indicated, the program should **default to fetch the data from the popular movies endpoint**

### 1.2.1.1 Starting the terminal spinner

Before making the network request, you will have to use the [ora](#) npm package to run a spinner in the terminal that renders the following message until the request has finished: `Fetching the movies data...`

### 1.2.1.2 Making the network request

The next step is to make a network request using the built in `https.request()` module from node.js. The request should be made to the following url passing in the page as a query param that you will have to use for pagination.

<https://api.themoviedb.org/3/movie/popular?page=1>

The URL should be created dynamically depending on the commands and options that the user executing the program has passed when running the application.

If the user has chosen the `get-movies` command with the `--popular` option and the `--page 2` flag, you will have to make a network request to the URL as such:

```
./moviedb.js get-movies --page 2 --popular
```

<https://api.themoviedb.org/3/movie/popular?page=2>

### 1.2.1.3 Handling errors when making the request

If the network request fails, you will have to use the `ora.fail()` method to render the error message. This means that you will have to handle and listen for any errors that might be returned from the `https.request()` method and stop the spinner with the error message as the text output.

### 1.2.1.4 Rendering the movies data

Once you have the data from the API, you will have to render the results in the terminal using `console.log` and the [chalk](#) npm package so that you can render text in colors in the terminal.

### 1.2.1.5 Page number for pagination

If the total number of pages is greater than the current page, you will have to render a text output similar to the following using **`console.log`** and the **`chalk.white()`** method:

```
-----  
Page: 1 of: 500
```

### 1.2.1.6 Movies data

Then, for each movie entry from the network response, you will have to render using **`console.log`** and **`chalk`** the following content:

- a line of dashes to separate the movies from each other
  - using `chalk.white`
- an empty line
  - using `chalk.white`

- Movie:
  - using chalk.white and an ``\n`` character to insert a line break
- ID:
  - using chalk.white to render the id of the movie
- Title:
  - using chalk.bold in blue color to render the title of the movie
- Release Date:
  - using chalk.white
  - the movie's release date
- empty line

#### 1.2.1.7 Ending the terminal spinner

Once all the data has been rendered to the terminal, you will have to execute the `ora.succeed()` method to stop the spinner with the following text:

- if the request was made to the popular movies endpoint:
  - ✓ Popular movies data loaded
- if the request was made to the popular movies endpoint:
  - ✓ Movies playing now data loaded

#### 1.2.1.8 Example output of the data

```
Movie:
ID: 475557
Title: Joker
Release Date: 2019-10-02

✓ Popular movies data loaded
```

## 1.2.2 Single Movie details

URL: <https://api.themoviedb.org/3/movie/:movieId>

In this part of the pill you will have to implement the **get-movie command** that you can specify when you execute the `moviedb.js` file. This should be done using the **commander** package and the `.command(...)` method it provides.

This command will be used to load the information of a single movie based on the **id** you can get after executing the **get-movie command** in the first step.

The **get-movie** command should have the following required fields:

- **description:** **Make a network request to fetch the data of a single movie**
  - the description of the command that will be shown in the help menu so that the users of the CLI program can know what the command is used for at all times
- **options**
  - **--id**
    - shorthand version: **-i**
    - required: **yes**
    - description: **"The id of the movie"**
    - this option should accept an input value that indicates the **id** of the movie that you want to get the details of
  - **--reviews**
    - shorthand version: **-r**
    - required: **no**
    - description: **"Fetch the reviews of the movie"**

With this command and options in place, you will have to implement the functionality of the `./moviedb` file so that it makes a network request and renders the results in the terminal using several formatting requirements.

**The CLI program should be used in the following way:**

- `./moviedb.js get-movie --id movieId`
  - With this command you should be able to fetch the data of the movie that has the id indicated
- `./moviedb.js get-movie --id movieId --reviews`
  - With this command you should be able to fetch the data of the movie that has the id indicated and the reviews of the particular movie

#### 1.2.2.1 Starting the terminal spinner

Before making the network request, you will have to use the [ora](#) npm package to run a spinner in the terminal that renders the following message until the request has finished: `Fetching the movie data...`

#### 1.2.2.2 Making the network request

The next step is to make a network request to the following url passing in the movie id in the URL.

[https://api.themoviedb.org/3/movie/:movie\\_id](https://api.themoviedb.org/3/movie/:movie_id)

If the `--reviews` flag is indicated in the CLI command, you should make the network request to the following URL:

[https://api.themoviedb.org/3/movie/:movie\\_id/reviews](https://api.themoviedb.org/3/movie/:movie_id/reviews)

The URL should be created dynamically depending on the id option that the user executing the program has passed when running the application.

If the user has chosen the `get-movie` command with the `--id` option and the `694919` value, you will have to make a network request to the URL as such:

```
./moviedb.js get-movie --id 694919
```

<https://api.themoviedb.org/3/movie/694919>

### Movie Reviews:

<https://api.themoviedb.org/3/movie/694919/reviews>

#### 1.2.2.3 Handling errors when making the request

If the network request fails, you will have to use the `ora.fail()` method to render the error message. This means that you will have to handle and listen for any errors that might be returned from the `https.request()` method and stop the spinner with the error message as the text output.

#### 1.2.2.4 Rendering the movie's data

Once you have the data from the API, you will have to render the results in the terminal using `console.log` and the [chalk](#) npm package so that you can render text in colors in the terminal.

#### 1.2.2.5 Movie data

Then, for each movie entry from the network response, you will have to render using `console.log` and chalk the following content:

- a line of dashes to separate the movies from each other prepended by an empty line
  - using chalk.white
- Movie:
  - using chalk.white and an ``\n`` character to insert a line break
- ID:
  - using chalk.white to render the id of the movie
- Title:
  - using chalk.bold in blue color to render the title of the movie
- Release Date:
  - using chalk.white to render the release date of the movie
- Runtime:
  - using chalk.white to render the runtime of the movie
- Vote Count:
  - using chalk.white to render the vote count of the movie
- Overview:
  - using chalk.white to render the overview of the movie
- empty line
- Genres:
  - Using chalk.white and an ``\n`` character to insert a line break
  - If the movie genres array has any contents (it's not empty) you will have to render the name of the genre using chalk.white.
  - If the genres array is empty, you will have to render the text using chalk.yellow:
    - The movie doesn't have a declared genre
- empty line



- Spoken Languages:
  - using `chalk.white` and an ``\n`` character to insert a line break
  - if the `movie.spoken_languages` is not empty, you will have to render each language in the array using `chalk.white`.
  - if the `movie.spoken_languages` array is empty, you will have to render the text ``The movie: ${movie.id} doesn't have any declared languages`` using `chalk.yellow`.

#### 1.2.2.6 Ending the terminal spinner

Once all the data has been rendered to the terminal, you will have to execute the `ora.succeed()` method to stop the spinner with the following text:

✓ Movie data loaded

#### 1.2.2.7 Example output of the data

```
- Fetching the movie data...(node:460) ExperimentalWarning:
  exports is an experimental feature. This feature could c
  e
  \ Fetching the movie data...
  -----

Movie:

ID: 694919
Title: Money Plane
Release Date: 2020-09-29
Runtime: 82
Vote Count: 20
Overview: A professional thief with $40 million in debt
  life on the line must commit one final heist - rob a fu
  e casino filled with the world's most dangerous criminal

Genres:

Action

English
✓ Movie data loaded
```

### 1.2.2.8 Movie Reviews data

If the `get-movie` command has been executed with the `--reviews` option you will have to render the reviews of the movie. Therefore, for each movie review entry from the network response, you will have to render using `console.log` and `chalk` the following content:

- If the `reviews` array is not empty, you will have to render the following:
  - If the `totalPages` is greater than the `page` you will have to render:
    - a line of dashes to separate the pages from each other using `chalk.white`
    - Page: using `chalk.white` to render the `page` of the review and of:  
using `chalk.white` to render the `totalPages`
  - a line of dashes to separate the pages from each other using `chalk.white`
  - empty line
  - Reviews:
    - Author:
      - using `chalk.bold` in blue color to render the author of the review
    - Content:
      - In this step, for each review in the `reviews` array, you will have to check if the length of `review.content` is greater than 400 characters. In that case, you will have to slice the string from the character 0 to the character 400 and concatenate a `. . .` string to show only part of everything and 3 points, as a summary only

- using chalk.white to render the reviewText
  - empty line
- If the reviews array is empty, you will have to render the following:
  - `The movie: \${movieID} doesn't have any reviews` using chalk.yellow.

#### 1.2.2.9 Ending the terminal spinner

Once all the data has been rendered to the terminal, you will have to execute the `ora.succeed()` method to stop the spinner with the following text:

✓ Movie reviews data loaded

#### 1.2.2.10 Example output of the data

```
| Fetching the movie data...
-----

Author: MohamedElsharkawy
Content: The Dilwale Dulhania Le Jayenge is a film considered
by most to be one of the greatest ever made. From The American
Film Institute to as voted by users on the Internet Movie Dat
abase (IMDB) it is consider to be one of the best.

✓ Movie reviews data loaded
```

## 1.3 Phase 3 (Extra step)

In this phase you will have to implement the `--save` and `--local` flags.

**These options will have to be implemented for the `get-persons` and `get-movies` commands only.**

Before beginning this part of the project, you will need to make sure to read the official documentation of the node.js fs module so that you can learn how to use it.

**You will have to use the callback version of each method in the fs module and not the Promises one.**

```
fs.readFile('/path', (err, data) => {  
  if (err) throw err;  
  console.log(data);  
});
```

### 1.3.1 `--save`

The `--save` flag will have to be added to each of the previous commands so that when it is included when executing the CLI program, instead of rendering the data in the terminal, it should store the contents in a **json file** inside a **files** folder.

You should check if the folder already exists before storing the files so that you don't try to save it in a folder that does not exist. In the case of the json files, you can overwrite the contents each time you save them, you don't have to append the data or check if it already exists each time.

Each type of information should be stored in a particular folder:

- the persons data in a **/persons** folder

- and the movies data in a **/movies** folder
  - the movies should be stored in a **popular-movies.json** file for the popular movies flag and a **now-playing-movies.json** file for the **--now-playing** flag

Once the data has been stored in the local file system, you will have to output a message in the terminal using the **ora** package both for the success and error states.

Furthermore, you will have to use the **node-notifier** package to output a notification in your computer's operating system.

For this, you will have to use the **notifier.notify()** method to output a notification title and message.

### 1.3.2 --local

Using this option you will have to read the data of the persons or the movies from the local file system instead of making a network request.

In case the file doesn't exist, for example: if calling the **get-movies** command and there is not file in the local file system with a movies entry, you will have to output a message using the **ora** package and also a notification using the **node notifier** package telling the user that the file does not exist.

Otherwise, you will have to render the data in the terminal using the same formatting options specified above and a notification using the **node notifier** package saying that the data has been loaded from the local file system.

## 1.4 Phase 4 (Extra step)

In this phase you will have to finish implementing the `--save` and `--local` commands for the rest of the commands and optional flags of the pill.

## 2. Pill Organization

It's important to document your own work in each pill or project you do. This way, you can have a summary of what you did in each project. So since we use a repository we have a **README.md** file which can be used to document how our application works. In the resources section there are some links about github guidelines to document your project.

## 3. Requirements

- You must use GIT. It is important that the indications and commits are explicit and concrete enough to be able to understand the changes without the need to require additional information as much as possible.
- Create a clear and orderly directory structure
- Both the code and the comments must be written in English
- Use the camelCase code style for defining variables and functions
- In the case of using HTML, never use inline styles
- In the case of using different programming languages always define the implementation in separate terms

- Remember that it is important to divide the tasks into several sub-tasks so that in this way you can associate each particular step of the construction with a specific commit
- Delete files that are not used or are not necessary to evaluate the project

## 4. Deliverables

To evaluate the project you will need the following deliveries:

- Repository with the code
- Postman collection with all the used endpoints in the pill
- You must create a correctly documented README file in the root directory of the project (see guidelines in **Resources**)

## 5. Resources

### 5.1. Static resources

- Oficial web page: <https://nodejs.org/en/docs/>
- W3schools: <https://www.w3schools.com/nodejs/>
- NodeJS Tutorial: <https://www.tutorialsteacher.com/nodejs/nodejs-tutorials>
- chalk: <https://www.npmjs.com/package/chalk>
- commander: <https://www.npmjs.com/package/commander>
- dotenv: <https://www.npmjs.com/package/dotenv>
- node-notifier: <https://www.npmjs.com/package/node-notifier>
- ora: <https://www.npmjs.com/package/ora/v/0.3.0>

- Sample guide for README:

<https://gist.github.com/PurpleBooth/109311bb0361f32d87a2>

## 5.2. Multimedia resources

### 5.2.1. NodeJS Course for beginners

In this video you will see a crash course of NodeJS

**Youtube List Link:**

[https://www.youtube.com/watch?v=BhvLizVL8\\_o](https://www.youtube.com/watch?v=BhvLizVL8_o)

### 5.2.2. NodeJS Crash Course

In this crash course you will explore Node.js fundamentals including modules such as path, url, fs, events and we will create an HTTP server from scratch.

<https://www.youtube.com/watch?v=fBNz5xF-Kx4>

### 5.2.3. Setup environment variables with Node.js + Dotenv

#### **Tutorial**

In this course you will learn environment variables to protect your tokens, passwords, db connection strings, etc. It's also great for having one reference to a variable that may change.

<https://www.youtube.com/watch?v=zDup0I2VGmk>