# Documentation for Password Manager

This is a documentation explaining the approach used for the various tasks.

## 1. Creating the Environment

1. **Directory Structure Creation**: A structured directory layout using the `mkdir` command is created. The two main directories created are:
    - `password-manager/src` for source files.
    - `password-manager/data` for data storage.
2. **Empty Source Files**: Three empty source files are created within the `src` directory using the `touch` command. These files are:
    - `initialize.sh`
    - `passwords.sh`
    - `utils.sh`
3. **Main Script Initialization**: The main script for this project is named `password_manager.sh`. It is created in the root directory, `password-manager`. This script includes a minimal structure with a `main` function. A bash here-document structure is used to insert the initial content directly into the file. The script ensures that it only creates the file if it does not already exist to prevent overwriting.
4. **Setting Executable Permissions**: Finally, the command, `chmod +x`,is used to make the `password_manager.sh` script executable, allowing it to be run as a stand-alone script.

## 2. Creating the Menu System

1. **Function Creation**: A function called `show_menu` is created to display the menu for the password manager. It takes no parameters The function prompts a user to choose an option from the menu. The menu options are
    - Add new password
    - Get password
    - List accounts
    - Delete a password
    - Change master password
    - Exit
2. **User Input Handling**: The function, `show_menu` utilizes a `read` command to capture the user's input and a `case` statement to handle the different menu options. For the start, Each option outputs a message indicating the selected choice and serves as a placeholder for further implementation. The options are further improved in later tasks below
3. **Exit Confirmation**: For the "Exit" option, the script prompts the user for confirmation before exiting. It employs a nested `case` statement to handle the confirmation response, allowing the user to either exit or return to the menu based on their input.

4. **Main Function Loop**: Inside the `main` function, an infinite loop (`while true; do ... done`) is implemented to continuously call the `show_menu` function. This ensures that the menu reappears after each user interaction until the user decides to exit the program.

5. **Including External Scripts**: External source files are added later.

# 3. Master Password Creation

## Create Initial Functions

In the previous section, we created the `initialize.sh` file. In this file, we created three function: `initialize`, `create_master_password`, and `check_master_password`. Each function takes no parameters.

1. **Function `initialize`**: This function checks if the file `data/.MASTER` exists.
    - If the file exists, it calls the `check_master_password` function.
    - If the file does not exist, it calls the `create_master_password` function.

2. **Function `create_master_password`**: This fuctions contains a placeholder functionality that echoes "Creating master password...". It will later be designed to prompt the user to create a master password.

3. **Function `check_master_password`**: This functions contains a placeholder functionality that echoes "Checking master password... It will be designed to verify the master password when a user is asked to provide a password to login. ".

## Modify `password_manager.sh` Script

The script in `password_manager.sh` is modified to incorporate the initialization functionality by sourcing the `initialize.sh` script and invoking the `initialize` function just above the while loop in the main function.

1. **Sourcing Initialization Functions**: Near the top of the script, the initialization functions are sourced using:

```
source src/initialize.sh
```

2. **Global Variable**: A global variable `MASTER_PASSWORD` is declared at the top of the script to store the master password for the session.

3. **Main Function Update**: The `initialize` function is called at the beginning of the `main` function to ensure the master password is set up or verified before proceeding.

## Implement `create_master_password`

**Function: `create_master_password`**

**Purpose**: Prompt the user to create a master password, verify the password, and securely store it in a hashed format.

**Approach**:

- **Parameter(s):** Function takes no parameters
- **User Prompt**: Informs the user about the need to create a master password.
- **Password Input (While Loop)**: Utilizes a `while true` loop to continuously prompt the user to enter and re-enter the master password until the inputs match.
- **Verification**: Compares the two entered passwords for consistency.
- **Success Condition**: If the passwords match, the function:
  - Confirms the match.
  - Assigns the verified password to a variable (`MASTER_PASSWORD`).
  - Generates a hashed password using `openssl passwd -6` with a randomly generated salt, using `openssl rand -base64 16`.
  - Stores the hashed password in a file (`data/.MASTER`).
- **Failure Condition**: If the passwords do not match, it prompts the user to try again.

## 4. Implementing Password Verification

Two functions, `get_salt` and `check_master_password`, are used to implement password verification.

**Function `get_salt`**

**Purpose:** Extract the salt from a formatted password hash.

**Approach:**

- **Parameter(s):** A formatted password hash, typically of the form generated by the `openssl passwd` command.
- **Parsing**: It uses the `cut` command to parse the formatted password hash. The input string is divided into fields separated by the dollar sign `$`. The salt is typically the third field in this format.
- **Output**: The extracted salt is then output.

---

**Function `check_master_password`**

**Purpose:** Verify the master password entered by the user against the stored hash. This authenticates the user to the program.

**Approach:**

- **Parameter(s):** Function takes no parameters
- **Reading Stored Hash**: The stored master password hash is read from the `data/.MASTER` file into a local variable `password_hash`.
- **Extracting Salt**: The `get_salt` function is used to extract the salt from `password_hash` into a local variable `password_salt`.
- **Verification Loop**: Using an infinite loop `while true; do ... done`
  - The user is repeatedly prompted to enter their master password.
  - The input is stored in a global variable `MASTER_PASSWORD`.
  - The `openssl passwd` command is used to hash the user's input with the extracted salt, creating `user_hash`.
  - This `user_hash` is compared with `password_hash`.

    - If they match, the loop breaks, indicating successful verification. Otherwise, an error message is shown, and the loop continues.
- **Return Value**: The function returns `0` upon successful verification.

## 5. Adding Functionality to Create and Store Passwords

In this section, we add functionality to create and store paswords in a file. The functions in this section include `generate_password`, `encrypt_password` and `new_password`.

**Function: `generate_password`**

**Purpose:** Generate a random password of length 24 using the command `openssl rand -base64 24`.

**Approach:**

- **Parameter(s):** Function takes no parameters.
- **User Inputs:** No user input
- **Password Construction:** A random password of length 24 is generated using the command `openssl rand -base64 24`.
- **Output:** Echoes the generated password.

### Password Encryption Function: `encrypt_password`

**Purpose:** Encrypt the generated password using a master password.

**Approach:**

- **Function Parameters:** `master_password` and `random_password`.
- **Encryption Process:** It uses the `openssl enc` command with the following options:
    - `-aes-256-cbc`,
    - `-pbkdf2 -a`, and
    - `iter 64000`.
- **Output:** The function echoes the encrypted password, which can be stored securely. This is base64 encoded due to the option `-a`

### New Password Creation Function: `new_password`

**Purpose:** Integrate the password generation and encryption processes, allowing the user to create and store new passwords.

**Approach:**

- **Function Parameter:** It takes the `master_password` as its only argument.
- **Directory Setup:** The function ensures that the directory for storing passwords (`data/passwords`) exists.
- **User Interaction Loop:** It uses a `while` loop to handle user prompts for the account name and checks for existing passwords. It asks the user to confirm the account name and checks if a password already exists for the account, offering the option to overwrite it.
- **Password Management:** It generates a new password using the `generate_password` function. It then encrypts the new password using the `encrypt_password` function.

- **Saving Encrypted Password:** The function saves the encrypted password to a file named after the account. It provides feedback to the user about the saved password and prompts them to return to the main menu.

## 6. Password Retrieval

### Password Decryption: `decrypt_password`

**Purpose:** Securely decrypt a password that has been previously encrypted using the `aes-256-cbc` cipher, base64 encoding, and a master password.

**Approach:**

1. **Arguments Handling:** The function accepts two arguments: `master_password` (the password used for decryption) and `ciphertext` (the base64 encoded encrypted password).
2. **Decryption Process:** The `openssl enc -d` command is employed for decryption. Options such as `-aes-256-cbc`, `-a`, `-pbkdf2`, and `-iter 64000` ensure the decryption process mirrors the encryption parameters, maintaining security and consistency. The master password is passed to the `-pass pass:` option for key derivation.
3. **Output:** The function outputs the decrypted password.

### Password Display: `display_password`

**Purpose:** Present the decrypted password to the user in a secure and temporary manner.

**Approach:**

1. **Arguments Handling:** It takes a single argument, `plaintext_password`, which is the password to be displayed.
2. **Displaying the Password:** The password is printed using `echo`. The user is prompted to press Enter when they are done viewing the password.
3. **Security Measure:** The `clear` command is used to remove the password from the terminal, preventing it from lingering on the screen.

### Password Retrieval: `retrieve_password`

**Purpose:** Manage the process of retrieving, decrypting, and displaying stored passwords for user accounts.

**Approach:**

1. **Arguments Handling:** It takes the `master_password` as its sole argument.
2. **Checking for Stored Passwords:** It checks if the `data/passwords` directory is empty using `ls -A`. If it is empty, it returns 1 and clears the screen.
3. **User Interaction Loop:** A `while` loop is implemented to prompt the user for an account name. It verifies if the account file exists in the `data/passwords` directory. If the account file does not exist, it offers the user an option to quit or continue the loop.
4. **Password Decryption and Display:** Once a valid account is selected, the encrypted password is read from the corresponding file. The `decrypt_password` function is called to decrypt the password. The decrypted password is passed to the `display_password` function for secure display.
5. **Exiting the Loop:** The loop breaks after successfully displaying the password.

**Some Improvements**

> Ensure your `decrypt_password` function handles errors gracefully, such as when the decryption fails due to an incorrect master password or corrupted ciphertext.

> Consider adding input validation to ensure the account name doesn't include any characters that could cause issues.

> You might want to log failed attempts to access non-existent account names for security auditing.

## 7. Listing Accounts

**Purpose:** List all stored account names within a specified directory (`data/passwords`). The function checks whether the directory is empty and, if not, displays the account names in a user-friendly format.

**Approach:**

1. **Function Definition and No Argument Requirement:** The `list_accounts` function is defined without any parameters, meaning it does not require any input arguments to execute.
2. **Directory Check for Emptiness:** The function checks if the target directory (`data/passwords`) is empty using the `ls -A` command, which lists all files (including hidden ones) in the directory. The `-z` flag in the conditional statement checks if the output of `ls -A` is an empty string, indicating that the directory is empty.
3. **Handling Empty Directory Case:** If the directory is empty, the function displays a message informing the user that there are no saved passwords and prompts them to add a password. The function then returns a status code of `1`, indicating that the operation was unsuccessful due to the lack of stored accounts.
4. **Listing Accounts:** If the directory is not empty, the function proceeds to list the account names stored in the `data/passwords` directory using the `ls -1` command. This command lists one file per line, providing a clear and readable output of account names. The function also includes decorative elements, such as headers, to enhance readability.
5. **User Prompt:** After listing the account names, the function prompts the user to press Enter to return to the main menu. This user interaction ensures that the script waits for user input before continuing, enhancing the overall user experience.

By adding these improvements, the document becomes more organized and clearer, making it easier to understand and follow.