Unit 3 of 8 V

Ask Learn



# Exercise - Analyze and document code using GitHub Copilot tools

20 minutes

GitHub Copilot can help you understand and document a codebase by generating explanations and documentation. In this exercise, you'll use GitHub Copilot to analyze and document the library application. You'll also create a GitHub repository for the library application and generate a README file for the repository using GitHub Copilot.

This exercise includes the following tasks:

- Use GitHub Copilot to explain the library application codebase.
- Use Visual Studio Code to create a GitHub repository for the library application.
- Use GitHub Copilot extensions to analyze the solution and create a README.md file for the repository.

### Examine the library application codebase

Before you create a code repository and README file, you should take some time to examine the codebase. It's important to understand the app architecture and key features before documenting the project. You should also run the application to ensure that it performs as expected.

## Use GitHub Copilot to explain the library application codebase

GitHub Copilot can help you to understand an unfamiliar codebase by generating explanations at the solution and feature levels.

Use the following steps to complete this section of the exercise:

- 1. Ensure that you have the AccelerateDevGitHubCopilot solution open in Visual Studio Code.
- 2. Open the Chat view, and then enter a prompt that uses GitHub Copilot's @workspace participant to generate a description of the project.

For example, enter the following prompt in the Chat view:

plaintext

@workspace Describe this project

3. Take a couple minutes to review the response.

You should see a response similar to the following description:

markdown

This project is a library management system implemented in C#. It consists of three main components:

- 1. \*\*Library.ApplicationCore\*\*: This is the core application library
  containing the main business logic, entities, enums, and interfaces.
  It includes:
  - Entities such as Book, Patron, Loan, etc.
- Enums like MembershipRenewalStatus, LoanReturnStatus, and LoanExtensionStatus.
  - Services like LoanService and PatronService.
- Interfaces like ILoanRepository, IPatronRepository,
   ILoanService, and IPatronService.
- 2. \*\*Library.Console\*\*: This is a console application that interacts with the user. It includes:
  - The main entry point in Program.cs.
  - Configuration settings in appSettings.json.
  - Console interaction logic in ConsoleApp.cs.
- 3. \*\*Library.Infrastructure\*\*: This library handles data persistence.
  It includes:
- JSON-based repositories like JsonPatronRepository and JsonLoanRepository.
  - Data loading and saving logic in JsonData.

Additionally, there are unit tests located in the UnitTests directory, which test the functionality of the core services using the NSubstitute mocking framework and xUnit.

The project uses .NET 8.0 and is configured to build and test using GitHub Actions as defined in the build—test.yml file. The solution file is AccelerateDevGitHubCopilot.sln, which includes all the projects and their configurations.

4. Use the Solution Explorer view to expand the project folders.

5. Locate and open the ConsoleApp.cs file.

The ConsoleApp.cs file is located in the src/Library.Console folder.

6. Enter a prompt in the Chat view that generates a description of the ConsoleApp class.

For example, enter the following prompt in the Chat view:

plaintext

@workspace #editor Describe the ConsoleApp class

7. Take a couple minutes to review the response.

You should see a response similar to the following description:

#### markdown

The `ConsoleApp` class in `ConsoleApp.cs` is responsible for managing the console-based user interface of the library management system. It handles user interactions, processes user inputs, and coordinates actions with the underlying services and repositories.

#### ### Fields

- `\_currentState`: Tracks the current state of the console application.
- `matchingPatrons`: A list of patrons that match a search query.
- `selectedPatronDetails`: Holds details of the currently selected patron.
- `selectedLoanDetails`: Holds details of the currently selected loan.
- `\_patronRepository`: An instance of `IPatronRepository` for accessing patron data.
- `\_loanRepository`: An instance of `ILoanRepository` for accessing loan data.
- `\_loanService`: An instance of `ILoanService` for performing loanrelated operations.
- `\_patronService`: An instance of `IPatronService` for performing patron-related operations.

#### ### Constructor

- `ConsoleApp(ILoanService loanService, IPatronService patronService, IPatronRepository patronRepository, ILoanRepository loanRepository)`: Initializes the `ConsoleApp` with the provided services and repositories.

#### ### Methods

- `Run()`: The main entry point for running the console application.
- `PatronSearch()`: Handles the logic for searching patrons.
- `ReadPatronName()`: Reads the patron's name from the console input.
- `PrintPatronsList(List<Patron> matchingPatrons)`: Prints a list of matching patrons to the console.

- `PatronSearchResults()`: Handles the logic for displaying and selecting from the search results.
- `ReadInputOptions(CommonActions options, out int optionNumber)`:
  Reads input options from the console.
- `WriteInputOptions(CommonActions options)`: Writes the available input options to the console.
- `PatronDetails()`: Displays details of the selected patron.
- `LoanDetails()`: Displays details of the selected loan and handles
  actions like extending or returning the loan.

The class uses a state machine approach to manage different states of the console application, such as searching for patrons, displaying search results, showing patron details, and handling loan details.

- 8. Open the Program.cs file.
- 9. Enter a prompt in the Chat view that generates a description of the Program.cs file.

For example, enter the following prompt in the Chat view:

plaintext

@workspace #editor Describe the Program.cs file

10. Take a couple minutes to review the response.

You should see a response similar to the following description:

#### markdown

The `Program.cs` file is the entry point for the console application. It sets up the dependency injection container, configures services, and starts the `ConsoleApp`.

#### 1. \*\*Dependency Injection Setup\*\*:

- Creates a `ServiceCollection` to register services.
- Builds a configuration object from the `appSettings.json` file.
- Registers various services and repositories with different lifetimes (singleton, scoped).

#### 2. \*\*Service Registration\*\*:

- Registers the repositories: JsonPatronRepository and JsonLoanRepository.
  - Registers the services: LoanService and PatronService.
  - Registers the JsonData class as a singleton.
  - Registers the ConsoleApp class as a singleton.

#### 3. \*\*Service Provider\*\*:

- Builds the service provider from the registered services.
- Retrieves the `ConsoleApp` service and runs it asynchronously.

This setup ensures that all dependencies are properly injected and configured before the application starts running.

- 11. Expand the Library.Infrastructure project folder.
- 12. Add the JsonData.cs, JsonLoanRepository.cs, and JsonPatronRepository.cs files to the Chat context.

Use the **Attach Context** button to add the files to the Chat context. You can also use a drag-and-drop operation to add the files to the Chat context.

13. Enter a prompt in the Chat view that generates an explanation of the data access classes.

For example, enter the following prompt in the Chat view:

plaintext

@workspace /explain Explain how the data access classes work

14. Take a couple minutes to review the response.

You should see a response similar to the following description:

#### markdown

The `JsonData` class in `JsonData.cs` is responsible for managing the loading and saving of data from JSON files. It handles various entities such as `Author`, `Book`, `BookItem`, Patron`, and `Loan`. The class uses the `IConfiguration` interface to read file paths from the configuration settings, ensuring that the correct JSON files are loaded.

The constructor of the `JsonData` class initializes the file paths for each entity by reading from the configuration section "JsonPaths". If the paths are not specified in the configuration, it defaults to specific paths within a "Json" directory.

The `EnsureDataLoaded` method checks if the data is already loaded and, if not, calls the `LoadData` method to load the data from the JSON files. The `LoadData` method reads the JSON files and deserializes them into the respective lists of entities.

The `SaveLoans` and `SavePatrons` methods handle saving the loan and patron data back to the JSON files. These methods ensure that only a subset of properties is saved, maintaining data integrity and consistency.

The class also provides methods to populate entities with related data. For example, `GetPopulatedPatrons` and `GetPopulatedPatron` methods populate patron entities with their associated loans.

Similarly, `GetPopulatedLoan` and `GetPopulatedBookItem` methods populate loan and book item entities with their related data.

The `LoadJson` and `SaveJson` methods are utility methods for reading from and writing to JSON files. They use the `JsonSerializer` class to handle the serialization and deserialization processes.

Overall, the `JsonData` class serves as a central point for managing the persistence of data in JSON format, ensuring that the application can load, save, and manipulate data efficiently.

15. Take a couple minutes to examine the JSON data files that are used to simulate library records.

The JSON data files are located in the src/Library.Console/Json folder.

The data files use ID properties to link entities. For example, a Loan object has a PatronId property that links to a Patron object with the same ID. The JSON files contain data for authors, books, book items, patrons, and loans.

#### ① Note

Notice that Author names, book titles, and patron names have been anonymized for the purposes of this guided project.

### Build and run the application

Running the application will help you understand the user interface, key features of the application, and how app components interact.

Use the following steps to complete this section of the exercise:

1. Ensure that you have the **Solution Explorer** view open.

The Solution Explorer view is not the same as the Explorer view. The Solution Explorer view uses project and solution files as "directory" nodes to display the structure of the solution.

2. To run the application, right-click **Library.Console**, select **Debug**, and then select **Start New Instance**.

If the **Debug** and **Start New Instance** options aren't displayed, ensure that you're using the Solution Explorer view and not the Explorer view.

The following steps guide you through a simple use case.

3. When prompted for a patron name, type One and then press Enter.

You should see a list of patrons that match the search query.

① Note

The application uses a case-sensitive search process.

4. At the "Input Options" prompt, type 2 and then press Enter.

Entering 2 selects the second patron in the list.

You should see the patron's name and membership status followed by book loan details.

5. At the "Input Options" prompt, type 1 and then press Enter.

Entering 1 selects the first book in the list.

You should see book details listed, including the due date and return status.

6. At the "Input Options" prompt, type **r** and then press Enter.

Entering **r** returns the book.

7. Verify that the message "Book was successfully returned." is displayed.

The message "Book was successfully returned." should be followed by the book details. Returned books are marked with Returned: True.

- 8. To begin a new search, type **s** and then press Enter.
- 9. When prompted for a patron name, type **One** and then press Enter.
- 10. At the "Input Options" prompt, type 2 and then press Enter.
- 11. Verify that first book loan is marked Returned: True.
- 12. At the "Input Options" prompt, type **q** and then press Enter.
- 13. Stop the debug session.

### Create the GitHub repository for your code

Creating the GitHub repository for your code will enable you to share your work with others and collaborate on the project.

① Note

You use your own GitHub account to create a private GitHub repository for the library application.

Use the following steps to complete this section of the exercise:

1. Open a browser window and navigate to your GitHub account.

The GitHub login page is: https://github.com/login

- 2. Sign in to your GitHub account.
- 3. Open your GitHub account menu, and then select **Your repositories**.
- 4. Switch to the Visual Studio Code window.
- 5. In Visual Studio Code, open the Source Control view.
- 6. Select Publish to GitHub.
- 7. Accept the default name for the repository (AccelerateDevGitHubCopilotSelect) and select **Publish to GitHub private repository**.
  - ① Note

If you're not signed in to GitHub in Visual Studio Code, you'll be prompted to sign in. Once you're signed in, authorize Visual Studio Code with the requested permissions.

- 8. In the Source Control view, enter a commit message, such as "Initial commit", and then select **Publish Branch**.
- 9. Notice that Visual Studio Code displays status messages during the publish process.

When the publish process is finished, you'll see a message informing you that your code was successfully published to the GitHub repository that you specified.

- 10. Switch to the browser window for your GitHub account.
- 11. Open the new AccelerateDevGitHubCopilot repository in your GitHub account.

If you don't see your AccelerateDevGitHubCopilot repository, refresh the page. If you still don't see the repository, try the following steps:

a. Switch to Visual Studio Code.

- b. Open your notifications (a notification was generated when the new repository was published).
- c. Select **Open on GitHub** to open your repository.
- 12. On the Code tab of your AccelerateDevGitHubCopilot repository, select Add a README.
- 13. In the README.md editor, type Coming soon and then select Commit changes.
- 14. In the Commit changes dialog, select Commit changes.
- 15. Switch to Visual Studio Code and ensure that the Source Control view is open.
- 16. Open the Views and More Actions menu, and then select Pull.

The Views and More Actions menu can be opened using the ellipsis in the top-right corner of the Source Control view.

- 17. Open the Explorer view (not Solution Explorer), and then expand the **AccelerateDevGitHubCopilot** folder.
- 18. Open the README.md file.

You should see the message "Coming soon".

You'll be using GitHub Copilot Chat to update your repository's README file in the next section of this exercise.

## Create the project documentation for the README file

The README file is an essential part of any GitHub repository. The README provides information based on the needs of the project, project contributors, and stakeholders.

For this guided project exercise, your README file should include the following sections:

- **Project Title**: A brief, clear title for the project.
- **Description**: A detailed explanation of what the project is and what it does.
- **Project Structure**: A breakdown of the project structure, including key folders and files.
- Key Classes and Interfaces: A list of key classes and interfaces in the project.
- Usage: Instructions on how to use the project, often including code examples.
- **License**: The license that the project is under.

In this section of the exercise, you'll use GitHub Copilot to create project documentation and add it to your README.md file.

Use the following steps to complete this section of the exercise:

- 1. Open the Chat view.
- 2. Use drag-and-drop operations to add files to the Chat view context.
- 3. To generate project documentation for your README file, enter the following prompt:

#### plaintext

@workspace Generate the contents of a README.md file for the code repository. Use "Library App" as the project title. The README file should include the following sections: Description, Project Structure, Key Classes and Interfaces, Usage, License. Format all sections as raw markdown. Use a bullet list with indents to represent the project structure. Do not include ".gitignore" or the ".github", "bin", and "obj" folders.

#### ① Note

Using multiple prompts, one for each section of the README file would produce more detailed results. A single prompt is used in this exercise to simplify the process.

4. Review the response to ensure each section is formatted as markdown.

#### ① Note

You can update sections individually to provide more detailed information or if they aren't formatted correctly. You can also copy GitHub Copilot's response to the README file and then make corrections directly in the markdown file.

5. Copy the suggested documentation sections to the README.md file.

#### markdown

#### # Library App

#### ## Description

Library App is a console-based application for managing library operations such as patron management, book loans, and membership renewals. It uses a JSON-based data storage system and provides various services to handle library functionalities.

#### ## Project Structure

- AccelerateDevGitHubCopilot.sln
- README.md

```
- src
  - `Library.ApplicationCore/`
    - `Entities/`
      - `Author.cs`
      - `Book.cs`
      - `BookItem.cs`
      - `Loan.cs`
      - `Patron.cs`
    - `Enums/`
      LoanExtensionStatus.cs`
      - `LoanReturnStatus.cs`
      - `MembershipRenewalStatus.cs`
    - `Interfaces/`
      - `IPatronRepository.cs`
      - `IPatronService.cs`
      - `ILoanRepository.cs`
      - `ILoanService.cs`
    - `Library.ApplicationCore.csproj`
    - `Services/`
      - `PatronService.cs`
      - `LoanService.cs`
  - `Library.Console/`
    - `appSettings.json`
    - `CommonActions.cs`
    - `ConsoleApp.cs`
    - `ConsoleState.cs`
    - `Library.Console.csproj`
    - `Program.cs`
  - `Library.Infrastructure/`
    - `Data/`
      - `JsonData.cs`
      - `JsonLoanRepository.cs`
      - `JsonPatronRepository.cs`
    - `Library.Infrastructure.csproj`
- tests
  - `UnitTests/`
    - `ApplicationCore/`
      - `LoanService/`
        - `ReturnLoan.cs`
        - `ExtendLoan.cs`
      - `PatronService/`
        - `RenewMembership.cs`
    - `UnitTests.csproj`
## Key Classes and Interfaces
- **Entities**
  - **Author**: Represents an author with properties like `Id` and
`Name`.
  - **Book**: Represents a book with properties like `Id`, `Title`,
`AuthorId`, `Genre`, `ImageName`, and `ISBN`.
  - **BookItem**: Represents a specific copy of a book with properties
like `Id`, `BookId`, `AcquisitionDate`, and `Condition`.
  - **Loan**: Represents a loan with properties like `Id`,
`BookItemId`, `PatronId`, `LoanDate`, `DueDate`, and `ReturnDate`.
  - **Patron**: Represents a library patron with properties like `Id`,
```

`Name`, `MembershipStart`, `MembershipEnd`, and `Loans`.

#### - \*\*Enums\*\*

- \*\*LoanExtensionStatus\*\*: Enum for loan extension statuses.
- \*\*LoanReturnStatus\*\*: Enum for loan return statuses.
- \*\*MembershipRenewalStatus\*\*: Enum for membership renewal statuses.

#### - \*\*Interfaces\*\*

- \*\*IPatronRepository\*\*: Interface for patron repository with methods like `GetPatron`, `SearchPatrons`, and `UpdatePatron`.
- \*\*IPatronService\*\*: Interface for patron service with method `RenewMembership`.
- \*\*ILoanRepository\*\*: Interface for loan repository with methods like `GetLoan` and `UpdateLoan`.
- \*\*ILoanService\*\*: Interface for loan service with methods like
  `ReturnLoan` and `ExtendLoan`.

#### - \*\*Services\*\*

- \*\*PatronService\*\*: Implements `IPatronService` to handle patronrelated operations.
- \*\*LoanService\*\*: Implements `ILoanService` to handle loan-related
  operations.

#### - \*\*Console\*\*

- \*\*ConsoleApp\*\*: Main console application class that handles user interactions and application flow.
- \*\*ConsoleState\*\*: Enum for different states of the console application.
- \*\*CommonActions\*\*: Enum for common actions in the console application.

#### - \*\*Infrastructure\*\*

- \*\*JsonData\*\*: Handles loading and saving data to JSON files.
- \*\*JsonLoanRepository\*\*: Implements `ILoanRepository` to manage loan data.
- \*\*JsonPatronRepository\*\*: Implements `IPatronRepository` to manage
  patron data.

#### ## Usage

- 1. Clone the repository.
- 2. Open the solution file AccelerateDevGitHubCopilot.sln in Visual Studio.
- 3. Build the solution to restore dependencies.
- 4. Run the `Library.Console` project to start the console application.
- 5. Follow the on-screen instructions to search for patrons, view patron details, extend loans, return books, and renew memberships.

#### ## License

This project is licensed under the MIT License.

- 6. Adjust the formatting manually if needed, and then save the updated README.md file.
- 7. Open Visual Studio Code's Source Control view.

8. To have GitHub Copilot generate a commit message, select the icon on the right side of the message box.

You should see a commit message similar to the following message:

chore: Update README.md with project description and usage instructions

- 9. Stage and Commit the file updates.
- 10. Sync (or Push) your updates to the GitHub repository.

## Check your work

To check your work, complete the following steps:

- 1. Open the GitHub repository for the AccelerateDevGitHubCopilot project.
- 2. Use the **Code** tab to review the updated README.md file.
- 3. Ensure that the project structure described in the README file aligns the folder structure of the repository.
- 4. Review the commit history and locate the commit message generated by GitHub Copilot.

## Next unit: Exercise - Develop code features using GitHub Copilot tools



Next >