Appcraft

Release 0.5.4

Dux Tecnologia, Thiago Costa Pereira

Mar 13, 2025

Contents

Welcome to the Appcraft documentation for version 0.5.4!

Appcraft is a modular framework designed to simplify software development by providing a structured and extensible architecture. It enables developers to create and manage complex applications with ease, ensuring scalability and maintainability.

This version includes all the essential features and modules to help you get started with AppCraft.

1 Getting Started

To get started with AppCraft, follow these steps:

1.1 Installation

To install AppCraft, simply run the following command:

```
pip install appcraft
```

1.2 Starting a New Project

To start a new project, follow these steps:

1. **Create a new directory for your project:**

   ```
   mkdir project_name
   cd project_name
   ```

2. Initialize the project using one of the following commands:

- **To initialize the project with the desired templates using the following command:**

  ```
  appcraft init <template_names>
  ```

- **Where <*template_names*> should be replaced by the names of the templates you want to use, separated by spaces. For example:**

  ```
  appcraft init logs locales flask_ui flask_api sqlalchemy
  ```

This will initialize your project with the specified templates. Make sure to separate each template name with a space.

- **Or, to list available templates**

```
appcraft list_templates
```

For more details on available templates, refer to the Templates Documentation.

2 Templates

In AppCraft, templates define the foundation of a **layered architecture**, ensuring modularity and independence. Each template provides a specific set of features that seamlessly integrate into a project.

All templates are **modular and independent**, allowing them to be combined as needed, making project construction **flexible and scalable**. Whether you require **logging, localization, web interfaces, or API handling**, you can mix and match templates without compromising the core structure of your application.

By leveraging AppCraft's template-based system, developers can streamline development while maintaining a **clean and maintainable architecture**.

2.1 Base Template

The **Base Template** is the foundational template of the framework, providing the backbone for the project's architecture in layers. It serves as the starting point for any application using the framework, offering a solid foundation and essential functionalities for efficient project management.

Key Features:

- **Simplified Management and Execution of EntryPoints**: The Base Template facilitates the setup and execution of various entry points in your project, allowing you to define and organize the core functions of your application.
- **Automatic Management and Installation of Missing Dependencies**: The template automatically handles the installation of dependencies, ensuring that your project has everything it needs to run correctly without requiring manual setup.
- **Error Handler**: It includes a robust error handling mechanism, allowing you to capture and manage exceptions appropriately, ensuring the stability and integrity of the system.
- **CLI UI Theme**: The Base Template includes a user interface theme for the CLI, providing a visually organized and consistent way to interact with the system, improving the user experience.
- **Customizable Message Printer**: The template offers a system for printing custom messages, allowing you to display different types of messages (such as success, warning, error, critical, title, and info) in a clear and formatted way, making communication with users easier during execution.

Benefits:

With these features, the Base Template establishes a solid and easy-to-use foundation for building and expanding projects, keeping them organized while offering great control over dependencies and user interaction.

2.2 Logs Template

The **Logs Template** provides a structured and efficient logging system, enhancing error tracking, debugging, and application monitoring. It ensures logs are properly stored, rotated, and formatted to maintain clear visibility of system events.

The **Logs Template** is seamlessly integrated into the framework's **Core**, operating automatically without requiring additional setup or manual imports. It ensures all logging functionality is readily available across the application.

Key Features

- **Structured Logging System:** Implements a well-organized logging mechanism that categorizes logs by severity levels, ensuring clarity in debugging and monitoring.
- **Log Rotation for Efficient Storage Management:** Automatically rotates log files based on time intervals or file size limits, preventing excessive log growth and improving performance.
- **Configurable Logging Levels:** Supports various logging levels (*DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*), allowing fine-grained control over log verbosity and filtering.
- **Multiple Output Formats:** Provides structured log formats, including plain text and JSON, making it adaptable for both human-readable logs and machine-parsed analytics.
- **Enhanced Debug Mode:** In development mode, integrates **structlog, better-exceptions, and rich** to provide visually improved, color-coded, and more readable terminal logs.

Benefits

- **Improved Error Tracking & Debugging:** Enables faster identification of issues by structuring logs in a clear and organized format.
- **Optimized Log Storage Management:** Automatic log rotation prevents excessive disk usage and keeps logs well-organized over time.
- **Seamless Framework Integration:** Operates directly within the framework, eliminating the need for additional configurations or external logging setups.
- **Developer-Friendly Debugging:** Provides enhanced, visually formatted logs in debug mode, making it easier to trace and understand errors.

The **Logs Template** delivers a powerful, automated logging solution that enhances application monitoring and debugging while ensuring logs remain structured, scalable, and easy to manage.

2.3 Locales Template

The **Locales Template** provides a comprehensive **internationalization (i18n) system**, enabling seamless multilingual support within applications. It ensures that messages, commands, and outputs are dynamically translated based on the user's locale, improving accessibility and usability across different languages. The **Locales Template** is fully integrated into the framework's **Core**, requiring no additional setup. It leverages a built-in translation system to dynamically load locale-specific resources and ensures smooth adaptation to multiple languages.

Key Features:

- **Dynamic Language Detection:** The template automatically detects the system's preferred language or allows users to specify their preferred locale for translations.
- **Message Translation with Gettext Support:** It leverages the **gettext** system for managing translations, supporting **.po and .mo files**, ensuring efficiency in message handling.

- **Automatic Compilation of Translation Files:** The template detects uncompiled translation files and **automatically compiles them**, ensuring that translations are always up to date.
- **CLI Message Translation:** The **Locales Template** intercepts standard print operations, ensuring that messages displayed in the command-line interface are automatically translated before being printed.
- **Seamless Framework Integration:** Designed to work natively within the framework, the template requires no external dependencies and follows a modular structure for efficient multilingual support.

Benefits:

- **Effortless Internationalization:** Automatically manages translations without requiring manual intervention, providing a smooth multilingual experience.
- **Consistent User Experience:** Ensures that users receive messages in their preferred language, improving accessibility and usability.
- **Automated Translation Handling:** Automatically compiles missing translation files, keeping them updated without additional configuration.
- **Integrated CLI Localization:** Command-line outputs are localized dynamically, reducing the need for hardcoded translations in scripts.

The **Locales Template** offers a fully integrated, scalable, and automated approach to internationalization, ensuring that applications can adapt efficiently to multiple languages without additional complexity.

2.4 Flask UI Template

The **Flask UI Template** provides an integrated **FlaskApp** that can function independently or in combination with the **Flask API** and **SQLAlchemy** templates if installed. It ensures a seamless integration within a **layered architecture**, maintaining modularity and flexibility in web application development.

This template introduces a well-structured **presentation layer** for UI endpoints under *presentation/web/ui*, offering built-in support and examples for constructing **Web UIs** efficiently.. It automatically sets up a **Flask application**, configures **routing** for **dynamic views, static views, and assets** and includes **error handling** for missing routes.

Additionally, if the **SQLAlchemy Template** is installed, the **Flask UI Template** will attempt to initialize the database connection, providing built-in support for ORM-based persistence. Otherwise, it gracefully operates without database integration, ensuring versatility in different application setups.

2.5 Flask API Template

The **Flask API Template** provides an integrated **FlaskApp** that can function independently or in combination with the **Flask API** and **SQLAlchemy** templates if installed. It ensures a seamless integration within a **layered architecture**, maintaining modularity and flexibility in web application development.

This template introduces a well-structured **presentation layer** for API endpoints under *presentation/web/api/v1*, offering built-in support and examples for constructing **RESTful APIs** efficiently.. It automatically sets up a **Flask application**, configures **routing** for **APIs** and includes **error handling** for missing routes.

This template is designed to manage **APIs**, offering built-in support and examples for constructing **RESTful APIs** efficiently. It automatically sets up a **Flask application**, configures **routing** and includes **error handling** for missing routes.

This template includes essential **API dependencies**, such as:

- **flask-login** (user authentication)
- **flask-restful** (simplified API routing)
- **flask-cors** (cross-origin resource sharing)
- **flask-compress** (response compression)

Additionally, if the **SQLAlchemy Template** is installed, the **Flask API Template** will attempt to initialize the database connection, providing built-in support for ORM-based persistence. Otherwise, it gracefully operates without database integration, ensuring versatility in different application setups.

2.6 Web Scraping Template

The **Web Scraping Template** provides a structured environment for efficiently extracting data from websites. It includes **pre-configured scripts** and essential libraries for handling web requests, parsing HTML, and automating interactions with web pages.

Key Features

- **Multiple Web Scraping Adapters:** The template includes an **adapter for each scraping library**, allowing flexibility in choosing the best approach for different use cases.
- **Standardized Architecture:** It provides an **abstract base class** for web scraping adapters, ensuring a **consistent and reusable** structure across different implementations.
- **Service Demonstrations:** It includes examples of **data extraction and storage services**, showcasing best practices for handling scraped data.
- **Included Dependencies**

  This template integrates powerful web scraping tools, such as:
  - **browser_manager** (headless browser management)
  - **scrapy** (high-level web scraping framework)
  - **selenium** (browser automation)
  - **requests & requests-html** (HTTP requests and dynamic content rendering)
  - **beautifulsoup4, lxml, pyquery** (HTML/XML parsing)
  - **fake-useragent** (randomized user agents for avoiding detection)
  - **retrying & tenacity** (automatic request retrying for failed attempts)

2.7 Git Template

The **Git Template** serves as a foundational setup for initializing a local Git repository, establishing a structured branching strategy tailored to various development environments. This template ensures a standardized workflow, facilitating seamless collaboration and efficient version management across different stages of development.▯

Key features:

- **Structured Branching Strategy:** The template introduces an organized branching model that aligns with semantic versioning and distinct development environments, including: - **Development (`dev`):** A branch dedicated to active development and integration of new features. - **Alpha (`alpha`):** An environment for early testing phases, allowing for initial feedback and iterative improvements. - **Beta (`beta`):** A pre-release stage focusing on refining features and addressing identified issues before the final release. - **Production (`production`):** The stable branch representing the live application, ensuring reliability and performance for end-users.

- **Initial Repository Setup:** Upon initialization, the template configures the local repository with the aforementioned branches, providing a clear pathway for code progression from development to production.
- **Environment-Specific Configurations:** Each branch is equipped with configurations pertinent to its respective environment, facilitating tailored testing, deployment, and monitoring processes.
- **Semantic Versioning Integration:** The branching strategy incorporates semantic versioning principles, promoting clarity in version management and release cycles.
- **GitAdapter Integration:** The template includes a **GitAdapter** component, designed to streamline interactions with Git. This adapter simplifies command executions, automates routine tasks, and provides a consistent interface for Git operations, enhancing productivity and reducing the potential for errors.

Benefits:

- **Consistent Workflow:** By adhering to a predefined branching strategy, teams can maintain a consistent workflow, reducing complexities associated with code integration and deployment.
- **Enhanced Collaboration:** Clear delineation of development stages allows team members to work concurrently on different aspects of the project without conflicts, streamlining collaboration efforts.
- **Improved Release Management:** Structured branches corresponding to specific environments enable systematic testing and deployment, enhancing the overall quality and reliability of releases.
- **Simplified Git Operations:** With the GitAdapter, developers can perform Git operations more efficiently, focusing on development tasks rather than manual version control management.

The **Git Template** serves as a robust framework for managing codebases, ensuring that development practices are standardized, efficient, and scalable across various environments.

2.8 GitHub Template

The **GitHub Template** extends the capabilities of the **Git Template** by integrating GitHub repository management, ensuring that project repositories are configured with essential metadata and remote origin settings. This template automates the process of creating repositories based on project configurations, simplifying repository initialization and remote synchronization.

Prerequisite: Git Template The **GitHub Template** builds upon the **Git Template**, leveraging its structured branching strategy and GitAdapter for seamless repository management. Before using the **GitHub Template**, ensure that the **Git Template** is applied to set up the local repository.

Key Features:

- **Automatic Repository Creation:** The template automates the creation of GitHub repositories using project-defined settings, ensuring consistency across multiple projects.
- **Project Metadata Integration:** It fetches the **project name** and **description** from the project configuration and applies them to the repository, providing a structured and informative repository setup.

- **Remote Origin Configuration:** The local Git repository is automatically linked to the corresponding GitHub repository, setting it as the **remote origin** to enable seamless push and pull operations.
- **Branch Structure Inheritance:** Since it extends the **Git Template**, all structured branching strategies (*dev*, *alpha*, *beta*, and *production*) are maintained, ensuring a well-defined development and release workflow.
- **GitAdapter Integration:** The **GitHub Template** leverages the **GitAdapter** to manage repository interactions with GitHub efficiently, simplifying operations like pushing changes, checking repository status, and handling authentication.

Benefits:

- **Automated GitHub Repository Setup:** Reduces manual configuration steps by automatically initializing and linking repositories to GitHub.
- **Consistent Project Metadata:** Ensures that repository details, such as name and description, are properly set based on project definitions.
- **Streamlined Remote Management:** Eliminates the need for manual *git remote add origin* commands, making repository synchronization effortless.
- **Standardized Version Control Workflow:** By inheriting the **Git Template**, it maintains a structured approach to branching and versioning across all repositories.

The **GitHub Template** provides an automated and standardized approach to managing repositories on GitHub, reducing manual configuration efforts and ensuring consistency across projects.

3 Layered Architecture

In this section, we introduce the **Layered Architecture** of the AppCraft framework. The architecture follows a clean and modular design, ensuring flexibility and scalability. The system is divided into multiple layers, each with distinct responsibilities, allowing for easy maintainability and extensibility.

The architecture layers are:

1. **Runner**: The entry points of the application. It initiates the process and calls the **Presentation** layer or **Service** layer.
2. **Presentation**: This layer handles the user interface and presentation logic. It interacts with the **Application** layer to present data to the user.
3. **Application**: Manages interactions between the **Presentation** layer and the **Domain** layer. It handles business logic and operations, such as calling models from the **Domain** and managing adapters and frameworks from the **Infrastructure**.
4. **Domain**: Represents the core business entities. This layer contains the application's data models and business rules.
5. **Infrastructure**: Includes all the external systems, frameworks, and adapters, such as databases, third-party services, file management, and more.

3.1 Runner Layer

Runners are the entry points of the application. It initiates the process and calls the **Presentation** layer or **Service** layer.

Aqui está uma seção **"What to Do in Runners"** no mesmo formato das outras:

What to Do in Runners

- **Bootstrap the Application:**
  The Runner is responsible for initializing the entire application. It should set up dependencies, load configurations, and start necessary services before the application starts handling requests.
- **Wire Up Dependencies:**
  Ensure that dependencies, such as repositories, services, and infrastructure components, are instantiated and injected properly. This avoids tight coupling and ensures dependency inversion is respected.
- **Load Environment Variables and Configurations:**
  The Runner should read and apply configuration settings from environment variables, configuration files, or a settings module to ensure flexibility and separation of concerns.
- **Register Application Components:**
  All necessary components, such as repositories, services, and external integrations, should be properly registered and made available to the application.
- **Handle Application Lifecycle Management:**
  Ensure proper shutdown procedures are in place, such as closing database connections, stopping background jobs, and releasing resources when the application stops.
- **Integrate Logging and Monitoring:**
  Set up logging configurations and monitoring tools to track application performance and diagnose issues effectively.

What to Avoid in Runners

While creating runners in the AppCraft framework, it's important to follow best practices to ensure the runners remain clean and focused on their intended purpose. Here are some things to avoid:

- **Avoid Using Print Statements for Debugging:**
  Runners should not be used for logging or debugging purposes, such as using *print()* statements. Print statements should be used in the Presentation layer when interacting with the user or displaying output. Logging, on the other hand, should be handled in the Service layer for better management and traceability of events.
- **Avoid Performing Complex Actions in Runners:**
  Runners should only handle the orchestration of tasks, such as calling the appropriate methods from the Presentation or Service layers. Avoid placing complex business logic, lengthy computations, or time-consuming actions directly in the runner methods. For example, avoid making database queries, handling large data processing, or interacting with external systems directly inside the runner. This should be delegated to services or other components that can handle the workload asynchronously or in a more appropriate place.

Types of runners

In the AppCraft framework, there are two types of runners:

1. **Main Runners:**
   These are the primary entry points of the application. They are located in the *runners/main* directory. These scripts are typically used for the main execution flow of the application.
2. **Auxiliary/Secondary Runners:**
   These runners serve secondary tasks and are located in the *runners/tools* directory. These scripts are typically used for secundary tasks that are not the main execution flow of the application.

To define a class as a runner, it must meet the following conditions:

- The class must be located in either *runners/main* or *runners/tools*.
- The class must inherit from *AppRunner*.
- The methods that are designated as runners should be decorated with the *@AppRunner.runner* decorator.

Example of a Runner

Here is an example of how to define a runner:

```python
from application.services.app_service import AppService
from infrastructure.framework.appcraft.core.app_runner import AppRunner
from infrastructure.memory.adapters.app_adapter import AppAdapter
from presentation.cli.app_cli_presentation import AppCLIPresentation


class AppRunner(AppRunner):
    @AppRunner.runner
    def start(self):
        app_adapter = AppAdapter()
        app_service = AppService(app_adapter=app_adapter)
        presentation = AppCLIPresentation(app_service=app_service)
        presentation.start()

    def non_runner1(self):
        # This method does not show in the runner.
        pass
```

Running Applications

To execute the **Main runners** within your project, use:

```
python run
```

This command will run the **main runner** of your project that are located in the *runners/main* folder.

If there are **multiple main runners**, you will be prompted to select the desired **filename, class, and method** in the *runners/main* directory.

If you want to know how to **execute a specific runner**, refer to the section *Executing a Specific Runner*.

Running Auxiliary Runner

To execute **Auxiliary Runner** in your project, use:

```
python run_tools
```

This command will run the **auxiliary runner** of your project that are located in the *runners/tools* folder.

If there are **multiple auxiliary runners**, you will be prompted to select the desired **filename, class, and method** in the *runners/tools* directory.

If you want to know how to **execute a specific runner**, refer to the section *Executing a Specific Runner*.

Executing a Specific Runner

If you want to execute a specific runner, you can use the following command:

```
python run file_name class_name method_name
```

- *file_name*: The name of the file containing the class runner.
- *class_name*: The name of the class runner.
- *method_name*: The name of the method within the class runner that you want to run.

If the class contains only one method, you can omit the *method_name*:

```
python run file_name class_name
```

If the file contains only one class, you can omit the *class_name*:

```
python run file_name method_name
```

Alternatively, if the file has only one class and one method, both can be omitted, and the runner will be selected automatically:

```
python run file_name
```

Or, if there is only one file in the project, you can omit the *file_name* entirely:

```
python run class_name method_name
```

If there is only one file and one class, you can also omit the *method_name*:

```
python run class_name
```

Finally, if there is only one file, one class, and one method, you can simply use:

```
python run method_name
```

Or, if everything is automatically determined (only one file, one class, and one method), you can just run:

```
python run
```

This system allows for flexible execution of your runners, making it easier to manage the different entry points in your project. You can target specific files, classes, or methods directly, depending on your needs.

3.2 Presentation Layer

The **Presentation Layer** is responsible for handling user interactions and presenting the data. It is designed to separate the user interface (UI) concerns from the core business logic and other layers of the application.

In this layer, you'll typically find different components responsible for displaying information to the user and capturing their inputs.

The Presentation Layer is mainly composed of the following:

1. **CLI Presentations:**
   Handle the command-line interface (CLI) interactions.

2. **UI Presentations:**
   Manage user interfaces (like web or graphical UIs), though this is an optional component depending on the type of application you're building.

Here's a refined section on "What to Avoid in Presentation" for your documentation:

What to Do in Presentation

- **Follow the Dependency Inversion Principle:**
  The Presentation Layer should depend on the **Application Layer**, not on **Infrastructure** or **Domain** directly. It should communicate only through services or use cases.
- **Use DTOs for Data Transfer:**
  Always use **DTOs (Data Transfer Objects)** to structure incoming and outgoing data. This ensures that the Presentation Layer does not work directly with **Domain Models**, reducing coupling.
- **Validate User Input:**
  Validate all incoming data before passing it to the **Application Layer**. This prevents invalid or incomplete data from reaching business logic and causing errors.
- **Keep Controllers Lightweight:**
  Controllers (or handlers) should only be responsible for handling requests, calling **Application Services**, and returning responses. Business logic should never be in controllers.
- **Implement Proper Error Handling:**
  Capture and format errors in the Presentation Layer before sending responses. Use proper HTTP status codes in APIs and meaningful messages in UI or CLI applications.
- **Ensure Idempotency in APIs:**
  If exposing an API, ensure that **POST, PUT, and DELETE** operations are idempotent when necessary, avoiding unintended duplicate operations.
- **Decouple from the Framework:**
  The Presentation Layer should not be tightly coupled to any specific framework. Keeping it loosely coupled makes it easier to change frameworks if needed.
- **Use a Consistent Response Format:**
  Define and follow a **structured response format** for all API endpoints. This makes it easier to integrate with clients.
- **Apply Authentication and Authorization:**
  Ensure that API endpoints and UI actions are properly secured using authentication and authorization mechanisms before calling business logic.

What to Avoid in Presentation

In the **Presentation Layer**, it is crucial to maintain a clear separation of concerns. Here are some common pitfalls to avoid:

- **Avoid Business Logic in Presentation Layer:**
  Do not include any business logic or complex decision-making processes in the Presentation Layer. This logic should reside in the **Application Layer** or the **Domain Layer**. The Presentation Layer should only be responsible for presenting data to the user and handling user input.
- **Avoid Direct Database Access:**
  Avoid direct interaction with the database in the Presentation Layer. Database access and data manipulation should be handled by the **Application Layer** via services or repositories.

- **Avoid Heavy Computation in Presentation Layer:**
  Avoid heavy computations in the **Presentation Layer**. The **Presentation Layer** should focus on displaying the data and not processing large datasets or performing calculations. These tasks should be delegated to the **Application** or **Domain** layers.
- **Avoid Side Effects in Presentation Layer:**
  The **Presentation Layer** should avoid making changes to the system state, such as saving data to the database or triggering external actions. Instead, it should communicate with the **Application Layer** to handle such side effects.
- **Avoid Logging and Debugging in Presentation Layer:**
  Avoid placing logging or debugging statements in the **Presentation Layer**. Logs should be handled at the **Service** or **Domain** levels, not in the UI or presentation code.
- **Avoid Tightly Coupled Components:**
  The **Presentation Layer** should not be tightly coupled to the specific implementation details of the **Domain** or **Infrastructure** layers. It should rely on abstractions such as interfaces or services to interact with the underlying layers.

This section ensures that the **Presentation Layer** stays focused on what it's meant to do---displaying data and handling user interaction---while keeping other responsibilities in the appropriate layers.

Example of a CLI Presentation

Below is an example of a class that demonstrates the **CLI Presentation** using the *AppCLIPresentation* class:

```python
from application.dtos.app_dto import AppDTO
from application.services.app_service import AppService
from infrastructure.framework.appcraft.utils.component_printer \
    import ComponentPrinter


class AppCLIPresentation:

    class Printer(ComponentPrinter):
        domain = "app"

        @classmethod
        def welcome(cls, app_name: str):
            message = cls.translate("Welcome to {app_name}")
            cls.title(message.format(app_name=app_name))

        @classmethod
        def app_info(cls, app: AppDTO):
            app_dict = app.to_dict()
            cls.title("App Informations")
            for name, value in app_dict.items():
                cls.info(name, end=": ")
                cls.print(value)

    def __init__(self, app_service: AppService) -> None:
        self.app_service = app_service

    def show_informations(self) -> None:
        app = self.app_service.get_app()
```

(continues on next page)

(continued from previous page)

```
        self.Printer.app_info(app)

    def start(self) -> None:
        app = self.app_service.get_app()
        self.Printer.welcome(app.name)
```

In this example:

- *AppCLIPresentation* is a class responsible for presenting the application information in a command-line interface.
- *Printer* is a nested class that extends *ComponentPrinter* and contains methods to display different pieces of information (like *welcome()* and *app_info()*).
- *show_informations()* calls the *app_info()* method to display detailed information about the app, such as name, version, environment, etc.
- *start()* displays a welcome message with the app's name.

This is just one of the ways to implement the Presentation Layer, and you can expand it based on your project's needs. The Presentation Layer can be used for both presenting data and handling interactions, with flexibility for your particular application.

3.3 Application Layer

The **Application Layer** is responsible for orchestrating business logic within the application. It sits between the **Presentation Layer** and the **Domain Layer**, handling the flow of data and ensuring that all interactions between the different layers are seamless. This layer often contains **Services**, **DTOs** (Data Transfer Objects), **Mappers**, and sometimes **Use Cases**.

The **Application Layer** coordinates the actions of the **Domain Layer** by interacting with models, repositories, and services, and it prepares the data to be sent to the **Presentation Layer**.

Aqui está a seção **"What to Do in Application"**, seguindo o mesmo formato da **"What to Avoid in Application"**:

What to Do in Application

In the **Application Layer**, it is essential to follow best practices to ensure the proper orchestration of business logic and system interactions. Here are some key principles to follow:

- **Orchestrate Business Logic, Do Not Implement It**: The **Application Layer** should coordinate interactions between the **Domain Layer** and external layers. Avoid implementing business rules here; instead, delegate them to the **Domain Layer**.
- **Use DTOs for Data Transfer**: Introduce **Data Transfer Objects (DTOs)** to shape the data exchanged between the **Application Layer** and **Presentation Layer**, ensuring separation of concerns and avoiding unnecessary dependencies on domain models.
- **Implement Application Services**: Keep application logic within well-defined **Application Services**, ensuring that use cases are clearly implemented and reusable. These services should act as intermediaries between the **Presentation Layer** and the **Domain Layer**.
- **Leverage Mappers for Data Transformation**: Use **Mappers** to convert data between **DTOs** and **Domain Models**. This ensures each layer remains independent and prevents unnecessary coupling.
- **Enforce Transaction Boundaries**: If an application service involves multiple repository calls, ensure that transactions are properly managed to maintain data consistency.

- **Keep Dependencies on External Layers Minimal**: The **Application Layer** should depend on the **Domain Layer** but should not have direct dependencies on external frameworks or infrastructure. Use dependency injection to manage dependencies effectively.
- **Ensure Idempotency for Application Services**: Application services should be designed to handle duplicate requests safely and prevent unintended side effects.
- **Write Unit and Integration Tests**: The **Application Layer** should be well-tested, covering different scenarios for service orchestration, data transformation, and interactions with the **Domain Layer**.

What to Avoid in Application

While working within the **Application Layer**, it's important to adhere to the following guidelines to ensure a clean and maintainable architecture. Here are some common practices to avoid:

- **Avoid Business Logic in Application Layer:**

  The **Application Layer** should not contain complex business logic. Business rules, validations, and domain-specific logic should reside in the **Domain Layer**. The Application Layer is responsible for orchestrating the flow of data and calls to other layers, not for implementing business rules.
- **Avoid Direct Interaction with Presentation Layer:**

  The **Application Layer** should never interact directly with the **Presentation Layer**. Its responsibility is to handle the orchestration of processes and provide data to be presented by the Presentation Layer. The Application Layer should not be concerned with rendering data or user-facing interactions.
- **Avoid Data Formatting or Rendering:**

  The **Application Layer** should not be responsible for formatting data, rendering views, or handling any user-facing interactions. This is the responsibility of the **Presentation Layer**. The Application Layer should focus on managing application flow and communicating with the **Domain Layer** and other necessary components.
- **Avoid Persistence Operations in Application Layer:**

  The **Application Layer** should not directly handle data persistence or database interactions. These tasks should be delegated to the **Domain Layer** or **Infrastructure Layer**, ensuring that the Application Layer remains focused on its orchestration role.
- **Avoid Tightly Coupled Components:**

  Avoid tightly coupling the **Application Layer** to the **Presentation Layer** or the **Infrastructure Layer**. The Application Layer should rely on abstractions and interfaces to communicate with other layers. This promotes flexibility, testability, and separation of concerns.

By adhering to these practices, the **Application Layer** remains focused on its core responsibility---managing application flow---while ensuring that other concerns, like business logic and presentation, are handled in the appropriate layers.

Example of a DTO

A **DTO** (Data Transfer Object) is a simple object used to transfer data between the **Application** and **Presentation Layers**. It is often used to simplify complex domain models and provide data in a form that is easy to transfer over a network or display in a user interface.

Here's an example of a DTO:

```python
class AppDTO:
    def __init__(
```

(continues on next page)

(continued from previous page)

```
        self, name: str, version: str,
        environment: str, debug_mode: bool
    ):
        self.name = name
        self.version = version
        self.environment = environment
        self.debug_mode = debug_mode

    def to_dict(self):
        return {
            "name": self.name,
            "version": self.version,
            "environment": self.environment,
            "debug_mode": self.debug_mode
        }
```

In this example, the *AppDTO* class is used to structure the application data in a way that can be easily transferred to the **Presentation Layer**.

Example of a Mapper

A **Mapper** is responsible for converting between different types of objects or data structures. In the context of the **Application Layer**, a mapper is used to convert **Domain** models to **DTOs** and vice versa.

Here is an example of a Mapper that converts a *Domain* model to a *DTO*:

```
from application.dtos.app_dto import AppDTO
from domain.models.app import App


class AppMapper:
    @staticmethod
    def to_dto(app: App):
        return AppDTO(
            name=app.name,
            version=app.version,
            environment=app.environment,
            debug_mode=app.debug_mode,
        )

    @staticmethod
    def to_domain(app_dto: AppDTO):
        return App(
            name=app_dto.name,
            version=app_dto.version,
            environment=app_dto.environment,
            debug_mode=app_dto.debug_mode
        )
```

In this example, the *AppMapper* class defines methods for converting between the **Domain** model *App* and the **DTO** *AppDTO*. This separation allows for cleaner code and better maintainability.

Example of a Service

A **Service** in the **Application Layer** is responsible for executing business logic and coordinating operations between the **Domain** and **Presentation** layers. A **Service** typically calls the **Domain Layer** to retrieve or manipulate data and then formats the data for the **Presentation Layer**.

Here's an example of an **Application Service**:

```python
from application.dtos.app_dto import AppDTO
from application.mappers.app_mapper import AppMapper
from domain.interfaces.adapters.app_adapter_interface import (
    AppAdapterInterface,
)


class AppService:
    def __init__(self, app_adapter: AppAdapterInterface):
        self.adapter = app_adapter

    def get_app(self) -> AppDTO:
        app = self.adapter.get_app()
        app_dto = AppMapper.to_dto(app)
        return app_dto
```

Aqui está a explicação ajustada para refletir corretamente o fluxo de dados no exemplo:

In this example, the *AppService* is responsible for retrieving application information through the *AppAdapterInterface*, which abstracts the infrastructure details. The retrieved *App* object belongs to the **Domain Layer**. To ensure a proper separation of concerns, the service uses the *AppMapper* to convert the domain model into an *AppDTO*, which is specifically designed for communication with the **Presentation Layer**. By doing so, *AppService* acts as an intermediary, orchestrating data transformation and enforcing business rules **between the Domain and Presentation layers**, while keeping the infrastructure details decoupled.

3.4 Domain Layer

The **Domain Layer** is the heart of your application, where the business logic and rules are defined. It contains the **models** that represent your business entities, and the **infrastructure interfaces** that define the contract for persisting and retrieving these entities. This layer is independent of any external concerns, such as databases or frameworks, and focuses solely on the core logic of the application.

The Domain Layer serves to encapsulate the business rules and logic, ensuring that the rest of the system can interact with it without knowing the underlying implementation details.

What to Do in Domain

In the **Domain Layer**, it is essential to follow best practices to ensure a clean, maintainable, and scalable architecture. Here are some key principles to follow:

- **Encapsulate Business Logic**: The Domain Layer should be the central place for business rules and domain logic. Keep this layer independent of external frameworks or infrastructure to ensure testability and reusability.
- **Use Rich Domain Models**: Prefer using rich domain models that encapsulate both data and behavior instead of relying only on anemic data structures. This helps enforce domain rules and improves maintainability.

- **Keep Domain Models Independent**: Avoid dependencies on infrastructure, frameworks, or database models. The domain models should represent the business concepts without being affected by external concerns.
- **Define Clear Boundaries**: Clearly define the boundaries of your domain entities and value objects. Ensure that changes to one entity do not inadvertently impact unrelated parts of the system.
- **Follow Interface-Based Design for Repositories**: Define repository interfaces in the Domain Layer, keeping the implementation details in the Infrastructure Layer. This abstraction allows easy substitution and testing.
- **Use Value Objects When Applicable**: Instead of using primitive types everywhere, consider using **Value Objects** for attributes that have specific business meaning, ensuring data consistency and enforcing constraints.
- **Ensure Immutability When Possible**: Make domain models immutable whenever possible to prevent unintended side effects and maintain a predictable state throughout the application.
- **Write Unit Tests for Domain Logic**: The Domain Layer should be highly testable. Write unit tests to verify business rules and domain logic independently from external dependencies.

What to Avoid in Domain

In the **Domain Layer**, it is crucial to maintain a clear separation of concerns. Here are some common pitfalls to avoid:

- **Business Logic in Other Layers**: Avoid placing business logic in the **Application Layer**, **Presentation Layer**, or **Infrastructure Layer**. The **Domain Layer** should be the place for business rules and models.
- **Direct Database Access**: Avoid having database-specific code or direct access to databases in the **Domain Layer**. Use repository interfaces to abstract away the database interaction.
- **External Dependencies**: The **Domain Layer** should not have dependencies on external frameworks, libraries, or infrastructure concerns. Keep it focused on the core business logic and rules.
- **UI Logic**: Avoid placing UI-related logic in the **Domain Layer**. The **Domain** should be agnostic to the user interface and should not know how data is presented to the user.
- **Global State**: The **Domain Layer** should not depend on or modify global state. The state should be encapsulated in objects and managed through domain logic.

Example of a Model

Here's an example of a **Model** in the **Domain Layer**, representing a **User**:

```python
# domain/models/user.py
class User:
    def __init__(self, user_id: int, name: str, email: str):
        self.user_id = user_id
        self.name = name
        self.email = email

    def change_name(self, new_name: str):
        self.name = new_name

    def change_email(self, new_email: str):
        self.email = new_email
```

(continues on next page)

(continued from previous page)

```python
    def __repr__(self):
        return f"User(id={self.user_id}, name={self.name}, email={self.email})"
```

In this example, the *User* class is a core business model that represents a user entity. It encapsulates the data and behaviors related to a user in the system.

Example of Repository Interface

Here's an example of a **Repository Interface** in the **Domain Layer**, defining methods for interacting with **User** data:

```python
# domain/repositories/user_repository_interface.py
from abc import ABC, abstractmethod
from domain.models.user import User

class UserRepositoryInterface(ABC):
    @abstractmethod
    def get_by_id(self, user_id: int) -> User:
        """Retrieve a user by ID"""
        pass

    @abstractmethod
    def create(self, user: User) -> None:
        """Create a new user"""
        pass

    @abstractmethod
    def update(self, user: User) -> None:
        """Update an existing user"""
        pass

    @abstractmethod
    def delete(self, user: User) -> None:
        """Delete a user"""
        pass
```

In this example, the *UserRepositoryInterface* defines the contract for repository operations such as creating, updating, and deleting users. This repository is abstract and doesn't deal with database specifics---it only specifies what actions can be performed on the **User** entity.

Example of Adapter Interface

This is an example of an **Adapter Interface** in the **Domain Layer**, defining a contract for retrieving **App** data:

```python
# domain/adapters/app_adapter_interface.py
from abc import ABC, abstractmethod

from domain.models.app import App


class AppAdapterInterface(ABC):
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    def get_app(self) -> App:
        pass
```

In this example, the *AppAdapterInterface* establishes a clear contract for accessing application-related data. This interface does not deal with infrastructure-specific details (such as databases, filesystem or external APIs); instead, it defines the expected behavior that any concrete adapter must implement. By using this abstraction, the **Application Layer** can interact with the infrastructure in a decoupled manner, ensuring better maintainability and testability.

3.5 Infrastructure Layer

Under construction

3.6 Application Flow

This document describes the request and response flow in the application, ensuring a clear separation of concerns and proper layer responsibilities.

Request Flow

1. **Runner (entrypoint)** → Instantiates the application layers.
2. **Presentation** → Receives user input, converts it into DTO + Filters.
3. **Mapper (Application)** → Converts DTO to Domain Model.
4. **Service (Application)** → Applies business rules, working with Domain Model.
5. **Repository (Application)** → Sends Domain Model or Filters to a Adapter.
6. **Adapter (Infrastructure)** → Interacts with the database and returns the Infrastructure Model.

Response Flow

1. **Adapter (Infrastructure)** → Returns the Infrastructure Model.
2. **Mapper (Infrastructure)** → Converts Infrastructure Model to Domain Model.
3. **Repository (Application)** → Returns the Domain Model.
4. **Service (Application)** → Applies business rules, returning the Domain Model.
5. **Mapper (Application)** → Converts Domain Model to DTO.
6. **Presentation** → Prepares the API response.

This architecture ensures full decoupling between layers, making the system more maintainable and testable.

4 Contributing

We appreciate your interest in contributing! You can contribute in three ways:

1. *Creating New Templates*.
2. *Updating Existing Templates*.
3. *Translating Templates*.

4.1 Creating New Templates

To create a new template, follow these steps:

1 **Clone the repository:**

```
git clone https://github.com/duxtec/appcraft.git
```

2 **Navigate to the repository folder:**

```
cd appcraft
```

3 **Run the command to create a new template:**

```
python appcraft/utils/template_creator.py <template_name>
```

This command will generate a new folder: ` `appcraft/templates/<template_name>/` `

4 **Inside this folder, update the `__init__.py` file:**

- Add a *description* to describe the template.
- **Activate the template by setting the class constant:**

```
active = True
```

- The *files/* folder inside contains the template's core files.

5 **Develop or Edit the Template Using a Temporary Project**
It is **not recommended** to edit templates directly inside the *appcraft/templates/<template_name>* folder. This approach has several limitations:

Why?

- **Dependency Issues**: Templates often depend on a base template or other templates, which are not recognized properly when editing them in isolation.
- **Linting Problems**: Linters and IDEs may fail to detect dependencies, leading to false errors and a poor development experience.
- **Risk of Breaking the Template Structure**: Direct modifications may lead to inconsistencies or missing files.

Solution: Use a Temporary Project To safely develop or edit a template, it is best to use a **Temporary Project**. This allows you to work inside a fully functional AppCraft-generated project, where all dependencies, configurations, and integrations work correctly.

Instructions for creating a temporary project can be found here: *Working in a temporary project*.

6 **Submit a pull request with the new template**!

4.2 Updating Existing Templates

Updating an existing template follows the same process as creating a new one. Simply **skip steps 3 and 4** and proceed with the rest of the instructions.

For detailed steps, refer to *Creating New Templates*.

4.3 Translating Templates

To translate templates into another language:

1 **Identify the template to translate**.

2 **Create a new language folder inside the template**:

```
appcraft/templates/<template_name>/other_files/locale/
├── en/  # English
├── pt/  # Portuguese
├── es/  # Spanish
```

3 **Translate relevant files** while maintaining consistency.

4 **Create a new demo project and test the translation**.

5 **Submit a pull request with the translated version**.

4.4 Why Use a Temporary Project?

When working with templates, you might be tempted to edit them directly inside the *appcraft/templates/* folder. However, this approach has several drawbacks:

- **Dependency Issues:** Templates often depend on a base template or other templates, which are not recognized properly when editing them in isolation.
- **Linting Problems:** Linters and IDEs may fail to recognize dependencies, leading to false errors and a poor development experience.
- **Risk of Breaking the Template Structure:** Direct modifications inside *appcraft/templates/* may lead to inconsistencies or missing files in the final template.

To avoid these issues, **the recommended approach is to use a Temporary Project**, which allows you to develop templates inside a fully functional AppCraft-generated project. This ensures that all dependencies, configurations, and integrations work correctly before saving the template.

4.5 Working in a temporary project

To work in a temporary project:

1. **Uninstall AppCraft (if installed):**

   ```
   pip uninstall appcraft
   ```

2. **Navigate to the cloned AppCraft repository:**

   ```
   cd appcraft
   ```

3. **Install the local version:**

   ```
   pip install -e .
   ```

4. **Create a new project:**

   ```
   appcraft init <template_name> [dependencies]
   ```

   Replace *<template_name>* with the template you're working on and specify any additional dependencies.

   The temporary project will be created in: ` appcraft/templates/temp/ `

This project mirrors an AppCraft-initialized project, allowing safe editing.

5. **After modifying or creating files in the temporary project, save the template:**

   ```
   python appcraft/utils/template_saver.py <template_name>
   ```

   Modify only one template at a time.

   If you need to modify another template, **save the current template first**, then repeat the process starting from *appcraft init*.

4.6 Ready to Contribute?

If you have any questions, feel free to open an issue or reach out to the maintainers.

Happy Coding!

5 Contact

5.1 Under construction

6 Versions

This documentation contains multiple versions. Select the version you want to view:

Available Versions:

- Latest Version (0.5.4).

7 Version Information

This documentation corresponds to AppCraft version 0.5.4. For other versions, refer to the version selection page.

8 Features in Version 0.5.4

- Easy project creation and setup.
- Initial architecture and structure for new projects.
- Introduction of templates:
  - Base
  - Logs
  - Locales
  - Flask UI
  - Flask API
  - Web Scrapping

For detailed information, explore the sections listed in the *Overview*.