Main

Flutter web Development Configuration File with Proxy Server

SUMMARY

Web Development Proxy Server and Config File

Author: Sydney Bao (@SydneyBao), Salem Iranloye (@SalemIranloye)

Go Link: flutter.dev/go/web-development-proxy-server

Created: 05/2025 / **Last updated:** 06/2025

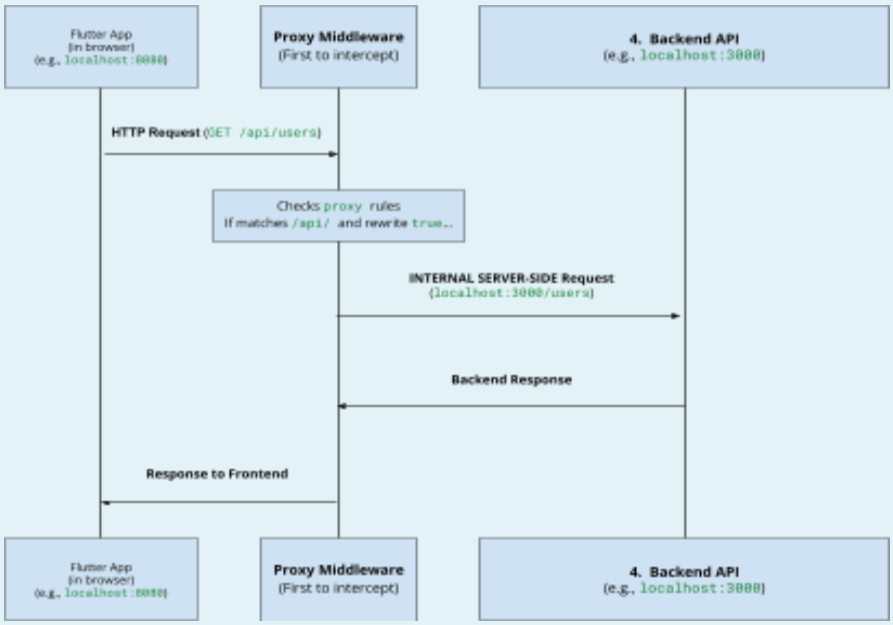
Related issues:

https://github.com/flutter/flutter/issues/51530

https://github.com/flutter/flutter/issues/117444

https://github.com/flutter/flutter/issues/67416

Diagram showing how requests are handled:



PUBLICLY SHARED

WHAT PROBLEM IS THIS SOLVING?

Flutter's current web development configuration relies on CLI arguments and does not have a development proxy.

Core Issues:

1. **Decentralized Application Hosting:** Hosting a Flutter application on a single server can be challenging, as there's no centralized gateway for API requests, forcing developers to hardcode API URLs in their frontend code and manage separate configurations for each backend service.
2. **Persistent CORS Errors:** Flutter applications can't directly call backend APIs without encountering Cross-Origin Resource Sharing (CORS) errors. This forces developers into workarounds or complex server-side configurations to resolve the issue.
3. **Discrepancy with Production Setups:** Unlike standard production environments where reverse proxies are common for traffic management and API security, Flutter lacks a built-in equivalent. This can lead to unforeseen issues and debugging challenges when applications are moved from development to deployment.
4. **Inconsistency with Industry Standards:** Developers coming from competing frameworks often expect a built-in proxy for local development. The absence of a built-in development proxy, a common and convenient feature in other frameworks, creates inconsistency with established industry practices, potentially hindering developer onboarding and workflow efficiency.
5. **Command-Line Configuration Overhead:** Developers often rely on command-line interface (CLI) arguments to manage web development settings. This approach lacks a centralized and easily configurable location, increasing the risk of inconsistencies across different development environments or team members.

BACKGROUND

This section outlines Flutter's current web development server configuration approach and contrasts it with solutions found in other prominent web development frameworks.

Flutter's Current Web Development Server

Currently, Flutter's web development server runs via `flutter run -d chrome` or `flutter run -d web-server`, and all web development settings are passed directly as CLI arguments.

Server Configuration and Proxying in Other Frameworks

Leading web development frameworks streamline the management of local development servers and proxies by centralizing configurations such as host, port, HTTPS, and proxy rules within dedicated files. This approach enhances both the robustness and developer-friendliness of the setup process.

- **Create React App (CRA):** Developers configure the `port`, `host`, and `HTTP` settings under the `start` field of `package.json`. They can add a `proxy` field for basic proxying. For more advanced needs, such as targeting multiple backend servers or

rewriting URL paths, developers can set up a `src/setupProxy.js` file. This file leverages the `http-proxy-middleware` package, offering granular control over proxy behavior.

- **Vue:** Developers handle the `host`, `port`, and `HTTPS` settings in `vue.config.js`. Proxy rules can be managed within the `devServer → proxy` field.

- **Vite:** `host`, `port`, and `HTTPS` settings are configured in `vite.config.js`. Developers can add a `server → proxy` field to manage proxy rules.

- **Angular:** Developers configure the `host`, `port`, and `HTTPS` settings in `angular.json`. Proxy rules are handled in a separate file, `proxy.config.json` (or `proxy.config.js`), which is then referenced when running `ng serve`.

Frameworks for Native/Hybrid Development (Different Context):

While not directly comparable to web frameworks due to different browser security models, it's worth noting how some native/hybrid frameworks handle API communication:

- **React Native:** Developers typically hardcode the IP address or hostname of the local backend server directly into their API client configurations (e.g., `Workspace('http://192.168.1.XX:3000')`). For complex local setups, developers might manually set up a custom Node.js proxy server.

Glossary

- **CLI (Command Line Interface) Command:** Text-based instructions for interacting with a computer system via the command line.
- **CORS (Cross-Origin Resource Sharing):** A browser security mechanism that prevents web pages from making requests to different origins.
- **CORS Error:** Browser-thrown error when a cross-origin request is blocked due to missing server permissions.
- **Development Proxy (or Proxy Server):** Local server in frontend development that forwards API requests, bypassing CORS during development.
- **Reverse Proxy:** A server that sits in front of backend servers, forwarding client requests, and is often used in production to avoid CORS issues by making requests appear same-origin.

OVERVIEW

The key features of this proposed solution include:

- **Dedicated Configuration File:** An optional configuration file, `web_dev_config.yaml`, will be introduced to centralize web-specific development settings. This file will enable the persistent and version-controllable definition of parameters such as headers, host, port, and HTTPS configurations. Unlike `pubspec.yaml`, which is utilized during the build process, this file is specifically used for development. Any settings provided via CLI arguments will take precedence over those defined in this file. The build halts only if the config file is empty or

missing the top-level `server` key, otherwise the error is logged and the web development settings revert to a default.

- **Integrated Development Proxy:** A built-in proxy allows developers to easily route API requests from their application through the local development server. This effectively resolves all CORS errors by making the application and its backend API appear as if they originated from the same origin. This feature simplifies the development process by enabling applications to be hosted on a single server during development, simulate the reverse proxy behavior common in production environments, and ensure alignment with the streamlined workflows found in competing web frameworks.

Non-goals

This document focuses on improving the configuration and proxying mechanisms for **Flutter web applications during development**. This proposal **does not** aim to:

- Remove CLI arguments
- Address production-level web server configurations or reverse proxy setups (e.g., Nginx, Caddy) that are typically handled at the deployment level.
- Introduce a full-fledged backend server within Flutter's development tooling; the proxy merely forwards requests to an existing backend.
- Resolve any issues related to embedding Flutter web applications within existing HTML pages or iframes, or the other way around.
- Implement solutions for advanced network debugging or traffic inspection tools beyond basic proxying.
- Hold environmental variables (there is already `--dart-define-from-file`)

USAGE EXAMPLES

This section illustrates how developers will interact with the new configuration system for Flutter web, demonstrating common use cases for defining server parameters and proxy rules, along with the specified configuration options.

The proposed `web_dev_config.yaml` file will organize these parameters under a `server` top-level key.

1. Simple Server Configuration and Proxying

Scenario: A developer wants to run the Flutter web app on a specific port and hostname, and proxy API calls to a local backend.

```
None
# web_dev_config.yaml

server:
```

```
port: 8080

host: "0.0.0.0"

proxy:

  - target: "http://localhost:5000/"

    source: "/users/"

  - target: "http://localhost:4000/"

    source: "/product/"

    replace: ""
```

Explanation:

- `Server → port` and `server → host` explicitly define where the development server will bind.
- The `server → proxy` section handles forwarding requests to the backend, bypassing CORS.

Flutter App Usage:

Java

```
// this is Dart source

final users = await http.get(Uri.parse('/users/names'));

// The dev server will forward the request to
   http://localhost:5000/users/names

final legacyItem = await
   http.get(Uri.parse('/products/item/123'));

// The dev server will forward the request to
   http://localhost:4000/products/item/123
```

Explanation:

- server → proxy defines multiple string proxy rules
  - `/users/` and `/products/` rules demonstrate direct proxying to different backend services (`localhost:5000` and `localhost:4000` respectively), effectively supporting a microservice architecture where each service has its own base path

    - The `replace: “”` field indicates that routes that match with `/product/` will remove the path `/product/` in the new url
- The Flutter application code remains clean, making relative API calls (e.g., `/users/profile`, `/api/users`). The development server's proxy transparently handles the complex routing and path transformations, ensuring that the browser's "same origin" policy is met and CORS issues are avoided during local development.

2. Advance Proxying

Scenario: A developer wants to run the Flutter web app with proxy API calls to a local backend using regexes and advanced replacement in the url.

```
None
# web_dev_config.yaml

server:

  port: 8080

  host: "0.0.0.0"

  proxy:

    - target: "http://localhost:5000/"

      regex: "/users/(\d+)/$"

    - target: "http://localhost:4000/"

      regex: "^/users/(\d+)/profile"

      replace: "/users/info"
```

Explanation**:**

- `server → proxy` defines regex proxy rules.
    - `/users/(\d+)/$` matches routes that fit this pattern exactly
    - `/users/(\d+)/profile` matches routes that start with this path and replaces that matches path with `/users/info`

3. Advanced Debugging and Browser Control

Scenario: A developer needs to specify TLS certificates for the dev server.

```
None
# web_dev_config.yaml
```

```
server:
  port: 8443
  host: "localhost"
  https:
    cert-path: "/path/to/dev_server.crt"
    cert-key-path: "/path/to/dev_server.key"
```

Explanation:

- `Server → https → cert-path` and `server → https → cert-key-path` configure the development server to use HTTPS with specified certificates.

4. Custom Headers

Scenario: A developer may want to inject custom HTTP headers into the development server's responses.

```
None
# web_dev_config.yaml
server:
  headers:
    - name: "X-Custom-Header"
      value: "MyValue"
    - name: "Cache-Control"
      value: "no-cache, no-store, must-revalidate"
  port: 8080
```

Explanation:

- `Server → headers` allows injecting custom HTTP headers into the responses served by the development server. This can be useful for various purposes, including setting security policies or debugging.

DETAILED DESIGN/DISCUSSION

Configuration

The proposed configuration for Flutter web development will reside in a `web_dev_config.yaml` file located in the root directory of the Flutter project. This file will contain rules for the web development server, including proxy configurations and general server parameters, organized in a nested structure.

An example `web_dev_config.yaml` reflecting the new structure might look like this:

```
None
server:
  headers:
    - name: "content-type"
      value: "application/json"
    - name: "X-Custom-Header"
      value: "my-value"
  host: "localhost"
  port: 8080
  https:
    cert-path: "/etc/ssl/certs/my_app.crt"
    cert-key-path: "/etc/ssl/private/my_app.key"
  proxy:
      - target: "http://localhost:3000"
        source: "/api/"
      - target: "https://auth.example.com"
        regex: "/auth/(\d+)"
```

Proxy Configuration

The `server → proxy` section is a list of **proxy configuration objects** where each entry has a **required** `target` field, the base URL of the backend server. Rules without a `target` field will be ignored. Requests will be matched based on the listed order of the objects.

Path

The path tells the server which requests to proxy. It can be one of two types:

- **String Prefix:** A simple path that matches requests with that string using the `source` field. Using a string prefix will match the beginning of a request. The `replace (optional)` field will replace the first occurrence of the matched path. To remove the matched path from the redirected route, set `replace` to `""` or `"/"`.
    - Example: `/api/`
- **Regular Expression:** For more flexible and complex matching use the `regex` field. `Replace` allows for capture groups (`$1`, `$2`, etc.). Expressions that end with a `$` will only match with identical routes. Expressions that start with a `^` will match the beginning of routes. `Replace` will replace all occurrences of the matched portion of the request exactly.
    - Example: `/user/\d+/`
    - Example: `/users/profile/` would reroute `/users/profile/summary` but `/users/profile/$` wouldn't
    - Example: `replace: /users/info` would reroute `/users/profilename` to `/users/infoname`

Examples

1. Basic String Proxy

Forwards requests from `/api/...` to a backend server. A request to `/api/v1/users` becomes `http://localhost:3000/api/v1/users`.

```
None
server:

  proxy:

    - target: "https://localhost:3000"

      source: "/api/"
```

2. String Proxy with Replace

Forwards requests from `/api/...` but removes the `/api/` prefix. A request to `/api/v1/users` becomes `http://localhost:3000/v1/users`.

```
None
server:

  proxy:

    - target: "https://localhost:3000"
```

```
      source: "/api/"
      replace: "/"
```

4. Advanced Regex No Replacement

Forwards requests from `/api/...` to a backend server. For instance, for a `user-id: 123`, a request to `/api/users/123` becomes `http://localhost:3000/api/users/123`.

```
None
server:
  proxy:
    - target: "https://localhost:3000"
      regex: "^api/users/(v\d+)"
```

3. Advanced Regex with Replacement

Remaps a versioned API path. A request to `/api/v1/users` becomes `http://localhost:3000/users?apiVersion=v1`.

```
None
server:
  proxy:
    - target: "https://localhost:3000"
      regex: "^api/(v\d+)/(.*)/"
      replace: "/$2?apiVersion=$1"
```

Implementation

The solution is integrated into the `flutter run` command for web, providing a seamless developer experience by loading a central configuration that dictates server behavior.

1. Configuration Loading on `flutter run`

The process begins when a developer executes `flutter run -d chrome` or `flutter run -d web-server`.

- **`loadDevConfig` Execution:** Before the web server is initialized, the `loadDevConfig` function is called to find and parse a `web_dev_config.yaml` file.

- **Configuration Merging:** `loadDevConfig` establishes a clear order of precedence for settings:
    1. **Command-Line Arguments:** Flags like `--web-hostname` or `--web-header` are given the highest priority.
    2. `web_dev_config.yaml` **File:** Values from this file are used if not overridden by a CLI argument.
    3. **Default Values:** A set of sensible defaults is used for any setting not defined in the other two locations.
- **`DevConfig` Instantiation:** The result of this process is a single, immutable `DevConfig` object that represents the final, consolidated configuration for this run session.

2. Configuration Data Flow

The `DevConfig` object is passed through the tool's core components to reach the server:

1. The `DevConfig` object is first passed to the `DebuggingOptions` data class.
2. `DebuggingOptions` then passes it to the `WebDevFS` constructor.
3. Finally, `WebDevFS` uses the `devConfig` object when it calls `WebAssetServer.start`, ensuring the server has all the necessary settings.

3. Data Models (`devfs_config.dart`)

All configuration is structured around a set of clear data classes:

- **`DevConfig`:** The primary class holding the entire server configuration, including host, port, a `list` of header objects, and a `list` of proxy rule objects.
- **`ProxyRule`:** An abstract class for proxy rules, implemented by:
    - `SourceProxyRule`: For simple path prefix matching.
    - `RegexProxyRule`: For advanced matching using regular expressions.
- **`HttpsConfig`:** Holds paths to TLS certificate and key files.

4. Web Server and Proxy Middleware

The implementation centers on refactoring `WebAssetServer` and using a `shelf` middleware pipeline.

- **Refactored `WebAssetServer.start`:** The method signature will be simplified to accept a single, required `devConfig` parameter, consolidating all web server settings into one object and making the code cleaner.
- **Proxy Middleware:** The proxy is implemented via `proxyMiddleware`. This middleware iterates through the `proxy` list from the `DevConfig` object. When an incoming request path matches a rule, in order listed in the yaml file, it forwards the request to the specified `target`, applying any replacement logic.
- **Request Handling Pipeline (`shelf.Cascade`):** The server defines a clear order of precedence for handling requests:
    1. **DWDS Handler:** The Dart Web Dev Service for debugging gets the first opportunity to handle the request.
    2. **Proxy Middleware:** If not handled, the request is passed to the `proxyMiddleware`.

3. **Asset Server Handler:** If no proxy rule matches, the request falls through to the main server to serve project assets like `index.html` and JavaScript files.

5. Error Handling and Logging

The system provides clear feedback to the developer:

- **File Not Found:** If `web_dev_config.yaml` is not found the server proceeds with defaults and CLI arguments.
- **Parsing Success:** When the file is loaded, the status is logged and the parsed content is logged when `--verbose`.
  Syntax Error: If the YAML file is malformed, the build is halted immediately, and a detailed error from the `package:yaml` parser is thrown, including the line, column, and problematic text to help the user fix it quickly.

ACCESSIBILITY

This proposed change, focusing on build-time configuration, has no impact on the accessibility features or behavior of the Flutter web application itself at runtime.

INTERNATIONALIZATION

This proposed change, being a build-time configuration, has no impact on the internationalization (i18n) capabilities or runtime behavior of the Flutter web application.

INTEGRATION WITH EXISTING FEATURES

The proposed configuration system is designed for seamless integration with existing Flutter web development workflows. The introduction of the `web_dev_config.yaml` file is **optional**, meaning current projects will remain unaffected unless a developer explicitly chooses to implement it.

This approach ensures backward compatibility and provides a clear opt-in path: developers can transition to using the new configuration by simply creating the `web_dev_config.yaml` file in their project's root directory. Furthermore, by aligning the configuration and proxying mechanisms with established patterns in other popular web frameworks, the learning curve for developers adopting this feature should be minimal, leading to a more intuitive and efficient transition.

TESTING PLAN

Unit Tests (`proxy_test.dart`):

- **ProxyRule.fromYaml**
  - Verify YAML configurations are parsed correctly and create appropriate `ProxyRule` subclass (`SourceProxyRule` or `RegexProxyRule`)
- **RegexProxyRule**
  - Verify **RegexProxyRule** instances are created correctly when a `regex` key is present, with and without a `replace` key

    - Verify `regex` paths are matched accurately
        - `^` matches the beginning of a request
        - `$` matches the request exactly
    - Ensure the `replace` method correctly applies replacements using capturing groups
        - Removes the `regex` path exactly from the redirected route
        - Removes **all** occurrences of the `regex` path if it doesn't start with `^`
- **SourceProxyRule**
    - Verify `SourceProxyRule` instances are created correctly when a `source` key is present, with and without a `replace` key
    - Verify `source` paths are matched accurately
        - Matches the beginning of a request only
    - Ensures the `replace` method works as expected
        - An empty string removes the path from the redirected route
        - Removes **first** occurrence of the `source` path
- **ProxyRequest**
    - Verify all essential elements of the original request are carried over to the proxied request
        - HTTP method (e.g., `POST`, `GET`)
        - Body content (including empty bodies)
        - Headers (excluding `Content-Length`)
        - Context information
    - Ensure the `url` of the proxied request is correctly updated to the `finalTargetUrl`
    - Ensure the `proxyRequest` correctly handles various HTTP methods (`GET`, `POST`, `PUT`, `DELETE`, `PATCH`)
- **ProxyMiddleware**
    - Verify that if no `ProxyRule` matches the incoming request's path, the middleware calls the `innerHandler`
    - Verify that if an exception occurs during the proxying process (e.g., due to an invalid target URL), the middleware gracefully falls through and calls the `innerHandler`
    - Ensure appropriate error messages are logged when a proxy error occurs

Integration Tests (demo):

Run `flutter run` with:

- No `web_dev_config.yaml` present → no proxying occurs, server starts normally.
- Empty `web_dev_config.yaml` → error is logged and build halts.
- Missing `server` top-level key → error is logged and build halts
- Missing `proxy.target` key → error is logged and rule is ignored. Build continues.
- `Headers`, `host`, `port`, `HTTPS`, and `proxy` are wrong types → Error is logged. Setting is ignored. Build continues.
- `web_dev_config.yaml` and a CLI argument are present → Overridden status is logged. CLI argument is used along with the other settings in the `web_dev_config.yaml`

DOCUMENTATION PLAN

This section outlines the plan for user-facing documentation to support the new Flutter web development server configuration. The documentation will be focused on practical usage and examples.

1. The `web_dev_config.yaml` Guide

This will be the main "getting started" guide. It will explain:

- **Purpose:** How this file provides a persistent, shareable configuration for the web dev server.
- **Location:** Where to create the file (`web_dev_config.yaml`).
- **Basic Structure:** How all settings must be configured and nested under a top-level `server:` key.

2. Basic Server Configuration

This section will detail the common server settings. For each setting, the documentation will explain its purpose and provide a syntax example.

- **`host` and `port`:** How to change the web server's binding address and port.
- **`headers`:** How to provide a list of global HTTP header objects to be injected on every response. The documentation will specifically show an example with a quoted value (like `Cache-Control`) to demonstrate handling of special characters.

3. Advanced Proxy Configuration

This will be the most detailed section, explaining how to solve CORS issues and route API requests. It will be broken down into clear concepts:

- **The `proxy` List:** Explains that `proxy` is a list where each **proxy configuration object** has a target and a path to match (either a string or regex).
- **Path:**
    - Describe how to use a simple **string prefix** using the `source` field for basic path matching (e.g., `/api/`).
    - Describe how to use a **regular expression** using `regex` filed for complex matching.
- **Proxy Behavior:**
    - Document the required `target` property for defining the backend URL.
    - Document the optional `replace` property, explaining the difference between using `replace` with:
        1. **Source:** replaces the first occurrence of a match
        2. **Regular expression:** For complex URL remapping using regex and capture groups. Replaces all occurrences.

4. Examples Gallery

To make the feature easy to adopt, the documentation will include a gallery of commented,

copy-paste-ready YAML examples for the most common use cases:

- A basic API proxy that forwards a path without changes.
- A proxy that strips the matched prefix from the URL.
- An advanced example that uses a regular expression and capture groups to remap a versioned API path to a query parameter.

5. Configuration Precedence

A small but critical section to prevent user confusion, explicitly stating the override order:

1. Command-Line Arguments (e.g., `--web-port`)
2. `web_dev_config.yaml` settings
3. Built-in default values

MIGRATION PLAN

Updating something that is not backwards compatible. Marking the CLI parameters as deprecated.

OPEN QUESTIONS

- How will maintenance overtime look?
- Should the `checked_yaml` package be added as a dependency to improve error messaging?
- While the errors are logged, should the build process halt for additional points of failure (e.g., Invalid setting types, proxy rules)?
- What parameters should be added to the config file?

FUTURE EXTENSIONS

- Would introducing devConfig flavors (e.g., `devconfig.local.yaml`, `devconfig.staging.yaml`), managed via the `--dev-config` CLI parameter (e.g., `flutter run -d chrome --dev-config devconfig.staging.yaml` be an effective way to manage different deployment environments?
- Should we support proxying WebSockets?
- Should we support glob syntax to define the path and replace rules within the development proxy?