ChatGPT

Connecting a Forge App to the Jira REST API (Backend Resolvers & requestJira)

Using the Forge API Module (`@forge/api`) for Jira REST Requests

Atlassian Forge provides the `@forge/api` module to call Jira Cloud REST APIs from your app's backend code. This module exposes an `api.fetch` interface and convenience methods like `api.asApp().requestJira()` and `api.asUser().requestJira()` for Jira REST calls [1] [2]. These methods automatically include the proper **Authorization** header for your app or user context, so you don't need to handle authentication tokens manually [3]. To use it, import the API and route utilities at the top of your code (e.g. `import api, { route } from '@forge/api';`) [4].

Context options:
- `api.asApp().requestJira()` – Makes the REST call as the app's *bot user* (app context). This is useful for elevated permissions or headless operations. As long as your Forge app has the required scopes, the request will succeed regardless of the end-user's permissions [1].
- `api.asUser().requestJira()` – Makes the REST call as the active user of the app. This respects the user's Jira permissions and may require the user to grant consent for certain scopes on first use [5]. If the user hasn't yet granted access, a 401 error triggers a consent prompt automatically [6]. (Note: asUser can only be used in modules where a user is present, like UI interactions, not in background triggers [7].)

Both methods return a Promise resolving to a Fetch API `Response` object. You can check `response.status` for success (e.g. 200 OK) and then use `await response.json()` (for JSON data) or `response.text()` to get the response body. By default, `requestJira` sends a GET request; you can specify other HTTP methods and payload via an `options` object (similar to `fetch` options) [8]. For example, to POST a change (like adding a watcher or updating an issue), include `{ method: 'POST', headers: {...}, body: '...' }` in the call [9].

> **Tip:** Use the `route` template literal from `@forge/api` to construct the URL path. The `route` function inserts the correct Jira site domain and encodes any variables safely [10]. For instance, `route\`/rest/api/3/issue/${issueKey}\`` will resolve to the proper REST URL for the issue in the site where the app is installed.

Defining Required OAuth Scopes in the Manifest

Forge apps must declare OAuth 2.0 scopes in the `manifest.yml` to gain access to Jira REST APIs. Scopes represent the permissions your app needs, and Jira will enforce both the scopes **and** the user's product

permissions. For a multiple-assignee tool that reads issue data and user info (and potentially updates issues), you will likely need:

- `read:jira-work` – to read Jira project and issue data (including searching issues, reading fields, etc.) [11].
- `write:jira-work` – to modify Jira issues (create or edit issues, update fields like assignee or add comments) [12].
- `read:jira-user` – to view user profiles and information in Jira (names, emails, avatars) [13].

Include these under the `permissions.scopes` section of your manifest. For example:

```
permissions:
  scopes:
    - read:jira-work
    - write:jira-work
    - read:jira-user
```

When an admin installs or upgrades your Forge app, they will see and grant these scopes. If your app tries to call an API without the proper scope, Jira will respond with **403 Forbidden** errors indicating missing OAuth permissions. Always double-check the Atlassian REST API docs for which scope is required for each endpoint (listed under "OAuth scopes required") [14]. In our case, the `/rest/api/3/issue` endpoints fall under Jira *work* data (read/write), and `/rest/api/3/user` falls under Jira *user* data, hence the scopes above cover those.

> **Note:** The app's scopes only request potential access. Jira's own project/issue permissions still apply and are not overridden by scopes [15] [16]. For example, if an app uses `asUser()` to fetch an issue but that user lacks *Browse Project* permission on that project, the call will still fail despite the app having `read:jira-work`. In such cases, using `asApp()` (with an app user that has broader access) may be necessary if appropriate for your app's design.

Example: Backend Resolver Calling Issue and User REST APIs

Using a Forge **resolver** (a backend function), you can fetch real Jira data and return it to your front-end. Below is an example of defining resolvers to get issue details and user information:

```
import Resolver from '@forge/resolver';
import api, { route } from '@forge/api';

const resolver = new Resolver();

// Resolver to fetch issue details by key or ID
resolver.define('getIssueDetails', async ({ payload, context }) => {
  const issueKey = payload.issueKey;
  // Call Jira API as the app (bot user) to retrieve the issue
  const response = await api.asApp().requestJira(route`/rest/api/3/issue/$
```

```
{issueKey}`);
  if (!response.ok) {
    console.error(`Failed to fetch issue ${issueKey}:`, response.status);
    throw new Error(`Cannot fetch issue ${issueKey}, status ${response.status}
`);
  }
  const issueData = await response.json();
  console.log(`Fetched issue ${issueKey}: ${issueData.fields.summary}`);
  return issueData;  // This will be sent back to the caller (frontend)
});

// Resolver to fetch a user's profile by accountId
resolver.define('getUserInfo', async ({ payload }) => {
  const accountId = payload.accountId;
  const response = await api.asApp().requestJira(route`/rest/api/3/user?
accountId=${accountId}`);
  if (!response.ok) {
    console.error(`Failed to fetch user ${accountId}:`, response.status);
    throw new Error(`Cannot fetch user ${accountId}, status ${response.status}
`);
  }
  const userData = await response.json();
  console.log(`Fetched user ${userData.displayName} (${accountId})`);
  return userData;
});

export const handler = resolver.getDefinitions();
```

In this example, we use `api.asApp()` to ensure the calls have the app's full permissions (so the data isn't limited by the viewing user). The use of `route` in `requestJira(route\` /rest/api/3/issue/${issueKey} `)` ensures the correct site URL is prepended and any special characters in issueKey are properly escaped [10]. The resolver functions log results with console.log` for debugging, then return the parsed JSON data.

Manifest setup for the resolver: In your `manifest.yml`, you need to register the function module and tie it to your Jira module (e.g., a UI extension). For instance, if you have a Jira Project Page or Custom UI app, your manifest might include:

```
modules:
  jira:customUI:
    - key: my-app-page
      resource: main
      resolver:
        function: my-app-resolver   # link to backend resolver
      title: My App
      # ... other module config ...
```

```
  function:
    - key: my-app-resolver
      handler: index.handler        # path to the resolver code export
      environments:
        - development
        - production

resources:
  - key: main
    path: static/my-app/build       # path to your frontend resources (for
custom UI)
```

The `resolver.function` field links the front-end module to the backend resolver. In development, ensure you deploy or run `forge tunnel` so that the resolver is registered; otherwise, calls to it won't be recognized. In the example above, the frontend can call `invoke('getIssueDetails', { issueKey: 'ABC-123' })` to execute the resolver and get data (see next section).

Frontend Requests: `requestJira` vs. Resolver Calls

Forge apps can call Jira APIs from the front-end as well, but there are important differences in how authentication is handled:

- **Custom UI or UI Kit 2 (Frontend):** The Forge bridge offers a `requestJira` function (in `@forge/bridge`) that allows the client code to call Jira REST APIs **as the current user** [17] [18]. For example, in a React component you might do:

```
import { requestJira } from '@forge/bridge';
const response = await requestJira('/rest/api/3/issue/ABC-123');
const issueJson = await response.json();
```

This will perform a GET to the issue endpoint using the logged-in user's credentials (scopes still apply). Under the hood, Forge injects the user's OAuth token, so you cannot change the auth context to the app in this direct call. If the user lacks permission for an operation (or hasn't consented to a scope), the request may be rejected or prompt for access.

- **Backend Resolver:** To use the app's identity (e.g. for admin-level actions or broader access), you must go through a resolver function. The front-end should call `invoke('<resolverName>', payload)` to have the backend run the request. In UI Kit or Custom UI, `invoke` is provided by `@forge/bridge` to call a named resolver. As Atlassian staff clarified, **front-end code can only make asUser calls**, and any call that needs app privileges must be done in a backend resolver using `api.asApp()` [19] [20]. In our example above, the UI would call `invoke('getUserInfo', { accountId })` to retrieve user details via the resolver (which uses the app context).

Why not call Jira directly from the browser? Forge imposes a content security policy that blocks direct AJAX calls to Jira Cloud from your iframe app. The correct approach is to use Forge's provided APIs. If you try to call `fetch('https://your-domain.atlassian.net/rest/api/3/...')` in the front-end, it will be

blocked (CSP error) [21] [22]. The `requestJira` bridge method avoids this by funneling the request through Atlassian's context, and prior to Forge Bridge v2.0 this always required a resolver proxy. Now, with bridge 2.0, Custom UI and UI Kit 2 apps can call `requestJira` directly from the front-end for user-context GETs and POSTs without the 10-second lambda limit [23]. However, if you need to perform an action with elevated privileges (using app scope or admin-only APIs), you still need to define a resolver and use `api.asApp()` on the backend [19].

Common Causes of Connection Errors (and Solutions)

When connecting a Forge app to Jira's REST API, developers often encounter permission or connectivity errors. Here are some frequent issues and how to resolve them:

- **Missing OAuth scopes:** If you receive 401/403 errors calling Jira APIs, verify that the required scopes are in your manifest and have been granted. For example, forgetting to add `read:jira-user` will cause calls to `/rest/api/3/user` to fail with authorization errors. Always check the endpoint's documentation for required scopes [14] and update your manifest accordingly, then re-deploy and re-install (or use `forge install --upgrade`).

- **User lacks permissions (when using asUser):** A call made with `api.asUser()` or front-end `requestJira` can fail with 403 **Forbidden** if the current user doesn't have the necessary Jira permission. For instance, a non-admin user calling an admin-only API (like project roles or user management) will get a permission error [24]. The solution is to either use `api.asApp()` in a resolver (with the app having an admin scope, e.g. `manage:jira-configuration`) [25], or ensure the user has the needed project role. Remember, asUser calls are constrained by the user's role and project access in Jira.

- **No resolver defined or not linked:** If your front-end invokes a resolver but nothing happens (or you see HTTP 409 errors in the console), it could mean the resolver function wasn't declared in the manifest or not deployed. Make sure your `jira:<module>` entry in manifest has a `resolver.function` pointing to a defined function module [26]. After adding a resolver, run `forge deploy` (or `forge tunnel`) again so it's active. A lingering error referencing a URL like `https://jira/rest/api/...` (with "jira" as host) is a sign that the call is not going through the Forge context properly [27] [10] – this usually traces back to a misconfigured resolver or using the wrong method to call the API.

- **CSP and incorrect API calls from front-end:** As mentioned, trying to call Jira REST from the client without using `requestJira` will violate content security policy [21]. Always use `requestJira` (for user-level calls) or a resolver (for app-level calls) instead of `fetch`/XHR in the UI code [22]. If you see a browser error about `connect-src` or CSP, double-check that you're using the Forge bridge correctly.

- **Forge environment context issues:** Ensure your app is installed in the correct Jira site and environment. For example, if you deploy to **development** and try using it in production, the API calls might fail. Use `forge install --environment development` for testing with `forge tunnel`, and install to production when you deploy there. Also, confirm that the app is running in a Jira

context (e.g., a Jira project page or issue panel). If a resolver is invoked outside of Jira (say, in Confluence by mistake), `requestJira` won't have a Jira site to target.

- **Exceeded execution time:** Forge resolver functions have a 10-second execution limit. If your resolver makes very slow Jira queries or many sequential calls, you might hit a timeout (which usually shows up as an error in the logs about the function terminating). In such cases, consider optimizing the query (e.g. fetch only needed fields or use Jira's bulk APIs), or use the front-end `requestJira` for long-running data fetches since those aren't subject to the 10s limit [28] [29].

By addressing the above, you can resolve most connection errors encountered when integrating with Jira's API.

Logging and Debugging Best Practices

When building a Forge app, effective logging and debugging are crucial:

- **Use `console.log` generously in the resolver:** You can log variable values, API responses, and error messages in your backend code. For complex objects, use `console.log(JSON.stringify(obj, null, 2))` to pretty-print them [30]. These logs will be captured by Forge and can be viewed with the CLI. After invoking your app, run the command `forge logs` (with `--environment` if needed) to retrieve recent log output [31]. This is the primary way to debug backend code in a deployed Forge environment. (During local development with `forge tunnel`, logs from the resolver are streamed live to your terminal.)

- **Inspect the response and errors:** Always check `response.status` and possibly the response body when calling the Jira API. If a call fails, do something like `console.error("Jira API error:", response.status, await response.text())` in your resolver. The error messages often contain clues (for example, a 403 might say "OAuth 2.0 scope missing" or "User does not have permission"). Logging these details to the console (and thus to Forge logs) will greatly speed up troubleshooting.

- **Front-end debugging:** Logs from your front-end code (UI kit or Custom UI) do **not** appear in `forge logs`. They must be viewed in the browser's developer console [32]. Use `console.log` in your React/JS code to verify that `invoke` calls are returning data, or to catch any exceptions on the client side. Also, if the UI is not behaving as expected, open the developer console to see if any errors (like network or JavaScript errors) are printed there.

- **Use Forge's debugging tools:** Forge provides a tunneling mode (`forge tunnel`) which is excellent for iterative development. In UI Kit 1, it live-reloads your app and shows logs immediately. In Custom UI, you can use `forge tunnel` for the backend while running your front-end locally, so you get real-time logging. Additionally, Atlassian's documentation suggests using the developer console and even IDE debuggers for more advanced needs (Forge supports attaching debuggers in certain scenarios) [33].

- **Check Atlassian Developer Console (if available):** When your app is installed, the Atlassian cloud site's admin section (**Apps -> Manage apps -> ...**) might show general error info if the app fails to

load. However, the detailed errors will still be in `forge logs` or the browser console. Always consult those logs for the true cause.

By combining these practices – logging in resolvers, examining HTTP statuses/messages, and using the Forge CLI tools – you can identify and fix issues quickly. Remember that any errors not caught in your code (e.g., an uncaught exception in a resolver) will also appear in the Forge logs with a stack trace, which can help pinpoint the problem.

Limitations and Considerations for Forge Jira API Calls

While Forge makes it straightforward to call Jira's APIs, be aware of some limitations and nuances:

- **Execution time limit:** As mentioned, backend Forge functions (resolvers, triggers) have a **10-second** execution limit by default. If your use case requires processing a large amount of data from Jira (for example, aggregating many issues or scanning all users), you may need to batch the work into multiple calls or offload some logic to the front-end. Forge's move to allow direct front-end `requestJira` calls helps with long-running fetches (no 10s limit on the client) [28], but long backend processes are still constrained. There is currently no way to extend the timeout beyond 10 seconds for a single invocation.

- **Rate limiting and throttling:** Jira Cloud's REST API has rate limits (both per user and per integration). A Forge app using `asApp()` has its own identity and may be subject to integration rate limits. In practice, the limits are generous for typical usage, but if your tool makes hundreds of requests in a short time, Jira may respond with HTTP 429 rate-limit errors. Design your app to use bulk APIs (like Jira's search endpoint or batch user lookup) to reduce call volume, and handle 429 responses with retries or backoff if needed.

- **App context permissions:** The Forge app's "bot" user generally has broad read/write access on the Jira instance for the scopes granted. However, it still must obey fundamental permissions schemes. For example, if a project has issue-level security or is private, the app might be treated as having default project access (Forge apps are typically added to the *atlassian-addons-project-access* role automatically). In rare cases, an admin might need to adjust permissions to ensure the app user can see the necessary projects or issues. This is usually handled by Jira automatically, but keep it in mind if your app cannot see data that it should (project permission schemes could be the culprit).

- **No user context in scheduled or event triggers:** If you use Forge **product events** (e.g., issue updated triggers) or scheduled tasks to call Jira APIs, note that those run without a user session. You cannot use `api.asUser()` in an event or schedule trigger unless you've set up an impersonation OAuth scope with `allowImpersonation: true` in the manifest [34]. Otherwise, you are limited to `api.asApp()` in those contexts. This is usually fine (the app can do what it needs), but it means any action that specifically requires a user identity (e.g., an audit as a user) might not be possible from a scheduled job without impersonation scopes.

- **Payload and response size limits:** Forge imposes some limits on payload sizes for functions and storage. While fetching typical issue or user data is fine, be cautious if retrieving very large datasets (like exporting all issues). The memory available to a Forge function is limited, and extremely large

JSON responses could cause the function to run out of memory or time. In such cases, consider filtering the REST call (using query parameters like `fields=` or pagination) to only retrieve what you need.

- **No direct external calls without whitelist:** Forge apps can only call the Atlassian product APIs (Jira, Confluence, etc.) using `requestJira` / `requestConfluence` by default. If your app needs to call **non-Atlassian APIs** (e.g., an external service), you must declare a remote in the manifest and use `api.fetch` with that remote. This doesn't affect Jira calls, but it's good to know you can't use `api.fetch` to call arbitrary URLs unless configured. Jira's own domain is implicitly allowed for `requestJira`, so no extra egress permission is required for our scenario.

- **Forge vs Connect differences:** Unlike Connect apps, Forge apps run in Atlassian's infrastructure. This means you don't have a long-running server to maintain state beyond each function invocation. Every call to a resolver is isolated (use Forge storage or state if you need to persist data between calls). Also, Forge logs and performance can be monitored via the developer console but require explicit retrieval as described above. The upside is authentication to Jira is handled for you (no need for OAuth dance or storing credentials), as demonstrated with `api.requestJira` in Forge.

In summary, a Forge app can reliably integrate with Jira's REST API for both reading and writing data, as long as you configure the correct scopes and adhere to the platform's constraints. By using backend resolvers with `@forge/api` for privileged operations and the front-end `requestJira` for user-level calls, you can build a powerful multiple-assignee tool that pulls real issue and user information. With careful attention to permissions, error handling, and logging, your deployed Forge app will be able to connect to Jira's API and provide live data in production.

---

1 2 3 4 5 6 7 8 9 Jira authentication
https://developer.atlassian.com/platform/forge/apis-reference/fetch-api-product.requestjira/

10 21 22 26 27 Forge API backend example help - Forge - The Atlassian Developer Community
https://community.developer.atlassian.com/t/forge-api-backend-example-help/82953

11 12 13 15 16 Jira product scopes
https://developer.atlassian.com/platform/forge/manifest-reference/scopes-product-jira/

14 19 20 24 25 Equivalent of api.asApp().requestJira with UI Kit2 - Forge - The Atlassian Developer Community
https://community.developer.atlassian.com/t/equivalent-of-api-asapp-requestjira-with-ui-kit2/77412

17 18 Forge bridge requestJira
https://developer.atlassian.com/platform/forge/apis-reference/ui-api-bridge/requestJira/

23 28 29 How to handle slow REST calls in Forge (10 seconds limit) - Forge Platform and Tools - The Atlassian Developer Community
https://community.developer.atlassian.com/t/how-to-handle-slow-rest-calls-in-forge-10-seconds-limit/47880

30 31 32 Debugging
https://developer.atlassian.com/platform/forge/debugging/

33 Find it, fix it, launch it: 5 Tips for debugging Forge apps - Atlassian
https://www.atlassian.com/blog/developer/find-it-fix-it-launch-it-5-tips-for-debugging-forge-apps

34 Permissions
https://developer.atlassian.com/platform/forge/manifest-reference/permissions/