ChatGPT

Surrogate Dependencies: Simulating Backends for Offline-First Development

By Dinis Cruz and ChatGPT Deep Research – August 2025

Introduction

Modern software development often relies on remote APIs and services, yet developers frequently need to work in environments where those backends are unavailable or unstable. **Surrogate dependencies** are a solution to this problem – they allow an application to run in a fully offline mode by simulating backend systems with prerecorded data. This concept was originally proposed as a way to align security and development needs, making developers more productive while still addressing AppSec concerns [1]. In essence, a surrogate dependency acts as a stand-in for a live API, serving consistent responses from local JSON data instead of making network calls. This technical white paper introduces surrogate dependencies, compares them to traditional stubs and mocks, and provides implementation strategies and examples for frontend applications (particularly those that normally communicate with FastAPI or other web services). We also discuss how surrogate dependencies support offline-first workflows, test-driven development (TDD), and even AI-assisted (LLM-based) development environments.

Background and Motivation

Applications that depend on cloud services or microservice APIs face several challenges during development and testing:

- **Fragile Dev/QA Environments:** When development or QA relies on live services, any downtime or change in those services can halt progress. Integration environments can be brittle and hard to maintain [2].
- **Limited Testing and TDD:** It's hard to do efficient TDD or integration testing when external dependencies are slow, unstable, or not under the developer's control [2]. Developers often end up writing fewer integration tests because running them requires a full environment.
- **Lack of Production-Like Data:** Using trivial stubs or dummy data means tests might miss issues that only real-world data would reveal [2]. Conversely, connecting to production or staging data can be risky or impractical, so teams lack realistic datasets in development.
- **Offline Development Constraints:** Without surrogates, developers cannot work offline (on an airplane or with poor internet) because the app won't function without its backend [2]. This dependency on connectivity slows down development and debugging.
- **Long Feedback Cycles:** Each backend call in a development environment can introduce significant latency. This slows down interactive work and automated test suites. The ability to run an application without live calls can remove these delays, speeding up the code–test–debug cycle [3].

The motivation behind surrogate dependencies is to resolve these issues by enabling applications to **run with zero external dependencies**. By capturing and simulating backend responses, developers get a fast, reliable "surrogate" for each dependency. Notably, this concept isn't purely about convenience – it also reflects a maturity in the development process. As Dinis Cruz notes, *"the ability to run your apps offline also signifies that the application development environment has matured to a level*

where you have… mocked versions of your dependencies" [4] . In other words, building offline capability via surrogate dependencies forces teams to design clearer contracts and test against them, benefiting both development and security. Ultimately, an environment where one can run everything locally (or in a contained setup) tends to produce more robust software, as it encourages comprehensive integration testing and rapid iteration [3] [5] .

What Are Surrogate Dependencies?

A **surrogate dependency** is a stand-in for a backend service that an application can use during development, testing, or even in demos, without needing the real service. Practically, a surrogate dependency consists of static data files (e.g. JSON files) and logic in the application to load those files in place of making a network request. The surrogate mimics the API's outputs: when the app "calls" the surrogate, it returns a predefined response identical in structure (and often content) to what the real API would provide. This allows the rest of the application to behave as if it were connected to the actual backend. As summarized in an OWASP talk, surrogate dependencies **"test the API and replay responses"** – developers use integration tests or recording tools to **lock in the API's behavior by saving responses in JSON format and then replay that data to the client**, thereby allowing the client to run entirely offline [6] .

It's useful to compare surrogate dependencies to related concepts:

- **Mocks/Stubs:** A *mock* or *stub* is typically a piece of code that imitates a real component by providing pre-set responses. For example, a stub function might always return the same hard-coded data. Mocks are often used in unit tests and usually require the developer to define expected outputs for each test scenario. Surrogate dependencies differ in that they use real data recorded from actual API calls, rather than hand-crafted responses. This makes surrogates more realistic and easier to scale across many endpoints – essentially, you are reusing real interactions instead of writing lots of stub code. Surrogates can be seen as "instance recordings" of a real service, whereas stubs are usually custom code implementations of an interface.
- **Service Virtualization:** In enterprise testing, service virtualization tools simulate the behavior of dependent systems (often with configurable rules, state, and more complex logic than a stub). Surrogate dependencies achieve a similar goal (isolating the app from the real backend) but in a far simpler way: by *playing back real responses*. You don't typically model the entire behavior or state of the service – you just serve static responses that were captured earlier. This is usually sufficient for front-end development purposes, where the concern is consuming the API, not modifying it. In essence, surrogate dependencies are a lightweight form of service virtualization, focusing on static snapshot data.
- **Caching (Evolved):** A surrogate dependency can be thought of as a **"more evolved cache."** In a traditional HTTP cache, responses from a server are stored and later served to avoid duplicate requests. Surrogates take this idea further: the cached responses are stored *explicitly and durably* (for example, checked into the code repository as JSON files), and the application is designed to use those cached responses **in place of** live calls during certain modes of operation. The goal isn't performance optimization (as in typical caching) but rather *full offline functionality*. In other words, a surrogate is a cache that completely replaces the live source when needed. By leveraging these recorded responses, developers ensure that "as much code as possible [is] running" without external dependencies [7] during development and testing.

By using surrogate dependencies, developers essentially decouple their front-end application from the backend API implementation. The application's network layer can be pointed either at the real API or at the surrogate data source. When pointed at the surrogate, the experience is seamless: from the application's perspective, it receives JSON data shaped exactly as if it came from the live service – except

it was loaded from a local store or CDN. This approach has been used in several projects to allow web apps to operate without any active backend, greatly enhancing testability and resilience in the development process.

How Surrogate Dependencies Work

1. Capturing Real API Responses: The first step is to gather real responses from the live backend. This is often done by writing integration tests or recorder scripts that exercise the real API endpoints. These tests invoke the API (e.g., making HTTP requests to the FastAPI service) and then save the resulting JSON responses to files. In Dinis Cruz's methodology, this process is described as using integration tests to "**'lock' the API**" – meaning you capture and lock-in the exact outputs of the API at a given time [6]. Each response is stored in a JSON file, typically organized in a structure mirroring the API routes (more on file organization in the next section). By committing these files to a repository (for example, a Git repo), the team creates a versioned, shareable dataset of what the API provided. An illustration of this capture process is shown below.

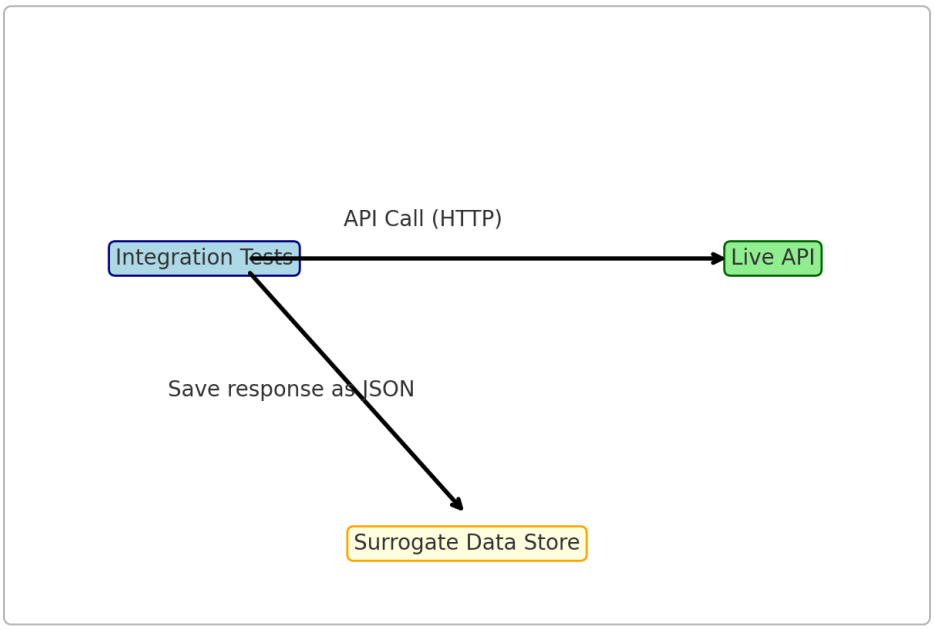

Figure 1: Integration tests call the live API and save its responses as JSON files in a surrogate data store. These files serve as the "surrogate dependencies" for offline use.

In this capture phase, the **application itself is not yet involved** – we are populating the surrogate data that the application will later consume. It's important to ensure the captured data covers the necessary use cases (all major endpoints, typical query parameters, etc.). Some teams capture not only normal responses but also error cases (e.g., a 404 or validation error response) so that the front-end can also handle those in offline mode. The capturing can be automated as part of a pipeline that regularly updates the surrogate data (for instance, after any API change or on a schedule) to avoid staleness.

2. Routing Application Requests to Surrogates: Once the JSON files are prepared, the application is configured to use them when in "surrogate mode." This typically involves a simple flag or mode switch in the application's configuration. The application's network layer (e.g., a fetch wrapper or API client module) checks this flag to decide whether to call the real API or to load a local file. In surrogate mode, what would normally be an HTTP GET to `https://api.example.com/items/42` might instead become a fetch of `surrogate_data/items/42.json` from the local bundle or a CDN. The surrogate

JSON contains the same fields the real API would return, so the rest of the application (UI logic, etc.) doesn't know the difference. The client effectively **replays the previously recorded responses** as if they were live [6]. The figure below illustrates an application using surrogate data in offline mode:

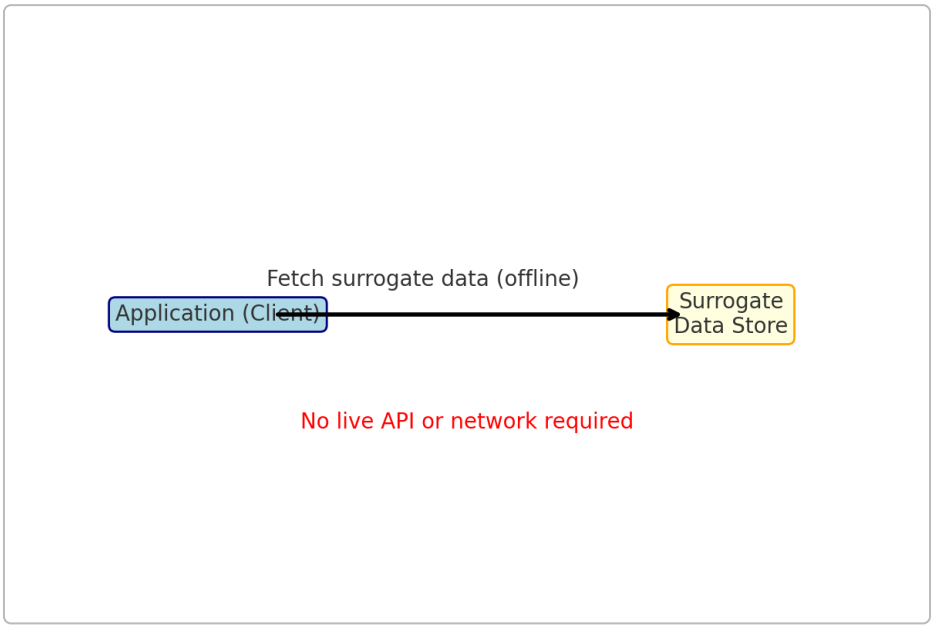

Figure 2: In offline mode, the application (client) fetches data from the surrogate store (pre-recorded JSON) instead of making calls to the live API. No network connection is required in this mode.

Crucially, all of this is achieved **without monkey-patching or hacky overrides** of low-level APIs at runtime. The ability to switch data sources is designed as part of the application architecture. For example, the app might have a central `apiClient` module or a function like `fetchData(endpoint)` that internally decides where to get the data. This design means that enabling or disabling surrogate mode is as simple as flipping a configuration, rather than modifying dozens of call sites or risking invasive patches to network libraries. The result is clean and maintainable: one code path for live mode, and one code path for surrogate mode, encapsulated in one place. In surrogate mode, developers can run the entire front-end application on their local machine (or even a static hosting environment) **with no internet connection**, and yet the app behaves almost exactly as it would online.

Benefits of Surrogate Dependencies

Surrogate dependencies bring a multitude of benefits to the development workflow:

- **Offline-First Development:** By allowing the app to run offline, surrogate dependencies remove the reliance on connectivity and backend availability. Developers can work from anywhere, and new team members can set up the project without needing special access to cloud environments. This offline capability is not just about convenience – it can significantly speed up development. For instance, remote API calls that might take seconds (or be altogether unavailable on an airplane) are replaced by local file reads that are virtually instantaneous. The overall feedback loop is faster. As one reference notes, *"The ability to run applications offline... is critical in the development process"* because it eliminates delays and enables faster, iterative coding [3]. Teams often find that with surrogates, they can adopt an **"offline-first"** mindset:

building the app to be fully functional in isolation, and only later wiring it to live services. This tends to result in more modular design and fewer assumptions about the environment.

- **Enhanced Test-Driven Development (TDD) & QA:** Surrogate dependencies make it feasible to run integration tests as part of the normal development cycle. Instead of tests that sporadically hit a staging server (which might be slow or have volatile data), tests can use the surrogate data to simulate those calls quickly and reliably. This encourages writing more integration tests (beyond unit tests) because the barrier of slow or flaky external calls is removed. In fact, capturing surrogate data can be integrated with TDD: a developer can write a test expecting a certain API output, run it in live mode once to capture the real output, save it, and then run all subsequent tests offline against that saved output. The surrogate effectively *locks in* the API's contract, so tests catch any deviation if the app's processing of that data changes [6]. Moreover, tests running against surrogates are deterministic and self-contained – a QA pipeline can run entirely offline, making continuous integration far less complex (no need to deploy a full test environment with all dependencies).

- **Realistic Data & Behavior:** Unlike trivial stubs, surrogate dependencies use real response data. This means the development and testing are done with JSON that has the exact schema, field names, and sometimes even production-like values (appropriately sanitized) from the real system. It's much more likely to uncover issues such as incorrect parsing, edge cases in data, or mismatches in expectations. For example, if an API sometimes returns an empty list or a null field, that exact scenario can be captured and reproduced offline, ensuring the front-end handles it. Developers gain a deeper understanding of the API's behavior by examining the surrogate JSON files. In a sense, the surrogate data set serves as documentation of the API. It answers questions like "what fields come in this response, and what are typical values?" without needing to constantly query the live service. This is particularly useful in LLM-based development environments where an AI assistant could be analyzing the JSON to understand the API (more on this later).

- **Faster Debugging and Troubleshooting:** When a bug appears in the interaction with a backend, having surrogate data means you can instantly reproduce the scenario without setting up a complex environment. You simply switch the app to surrogate mode (if it isn't already), load the relevant JSON (which might represent the problematic response), and step through the front-end code. There's no waiting for network calls, and you can even modify the JSON locally to test hypotheses (what if the API returned X instead of Y?). This ability to tweak inputs easily is akin to having a controllable test harness for your app's external interactions. It accelerates debugging dramatically.

- **Developer/Security Alignment:** Surrogate dependencies were partly born from an AppSec perspective – to give developers the tools to simulate production-like scenarios in a safe, controlled way [1]. By doing so, security testing can be baked in early. For instance, additional integration tests (or even fuzz tests) can run against the surrogate setup without risking hitting a real server with dangerous payloads. The OWASP slides suggest inserting security tests during the integration test phase (e.g., injecting malicious payloads through the surrogate mechanism) [8]. This means developers can verify how the application would react to certain API responses or errors (including security-relevant ones) offline. In summary, surrogates help create a common ground where both dev and security teams can experiment freely, leading to more secure and robust code.

- **Accelerating Collaboration and Onboarding:** When every developer has a local surrogate for the backend, the "works on my machine" problem is reduced. Environment setup becomes

easier – one doesn't need VPN access, database dumps, or special config to start working on the front-end. New developers or contributors can start the app with surrogate mode and see a fully functional UI immediately, which is great for understanding the project. Similarly, demo environments for stakeholders can use surrogates to show the product without needing live data (which might be sensitive or constantly changing). Because surrogate data sets are version controlled, specific scenarios can be shared: for example, a certain branch of the repository might include a surrogate dataset that demonstrates a new feature or a bug case. This improves communication within the team.

In summary, surrogate dependencies support a development culture where one asks *"Can you run your app offline?"* – a question that is increasingly a measure of the quality of the development environment [9]. The teams that can confidently answer "yes" tend to have faster release cycles and fewer integration surprises, whereas those that cannot often struggle with integration issues late in the cycle [10].

Challenges and Considerations

While surrogate dependencies bring many benefits, they also introduce certain challenges and trade-offs that teams should be aware of:

- **Data Freshness and Maintenance:** The surrogate JSON files represent a snapshot in time of the API's responses. If the real API changes (new fields, changed data formats, etc.), the surrogate data must be updated to match. Otherwise, the front-end might work in surrogate mode but break against the real API, or vice versa. Keeping surrogate data in sync requires a maintenance process. Strategies to address this include automating the data capture regularly (e.g., nightly builds that refresh the surrogates from a staging environment) and writing tests that compare surrogate outputs to live outputs to detect drift. The key is to treat the surrogate data as an extension of the codebase – it needs versioning, reviews, and updates alongside code changes. In practice, teams often find that API contracts don't change *too* frequently, and when they do, it's straightforward to re-run the capture scripts. But it's an ongoing responsibility nonetheless.

- **Coverage of Scenarios:** A surrogate will only have responses for the scenarios you've recorded. This is usually fine for standard use cases (e.g., the list of products, a sample product detail, etc.), but edge cases can be missed. For example, if the app can load user profiles by ID, you might record two or three user IDs' data. But what about a user with no profile picture, or one with an extremely long name, etc.? If such a case isn't in the surrogate data, the app in surrogate mode might never encounter it – possibly hiding a bug that would appear with a certain real user. To mitigate this, you should aim to capture a variety of representative data. It can also help to *generate* some surrogate data artificially for edge cases (essentially augmenting real data with synthetic cases). However, generating data veers towards traditional mocking. A balanced approach is to capture as much real variety as possible, and be aware of gaps. Also, the surrogate mode should not be the *only* testing mode – one should still test against the live API (at least in a staging environment) before production, to ensure nothing was missed. *"Of course you will always test against those other systems... If the dependencies don't pass [the same tests], you have a problem and you have to fix it,"* as noted in SecDevOps guidance [11]. In short, surrogate mode is a powerful tool, but it doesn't eliminate the need for final integration testing with real systems.

- **Data Volume and Size:** Storing JSON files for every needed endpoint and scenario can become unwieldy if the API is large. For instance, imagine an API with hundreds of endpoints – capturing

complete responses for all of them might result in thousands of JSON files, possibly large in size. This can bloat the repository or static asset package. Teams should be pragmatic in deciding what needs to be included. Often, a subset of endpoints (the most critical ones) are enough to enable most development tasks offline. Additionally, large data sets (like thousands of records) might be trimmed in the surrogate to a smaller representative set to keep things light. Remember, the goal is not to replicate the entire database, just enough data to develop and test the UI logic. Techniques like compression or hosting the surrogate data on a CDN (so it's fetched on demand rather than stored in the dev repo) can help if size becomes an issue.

- **Security and Privacy:** If you capture real responses from a production system to use as surrogates, be cautious about sensitive information. Those JSON files could contain user data, API keys, or other confidential info. It's often better to capture from a staging environment or sanitize the data before saving. For example, replace any real user emails or IDs with dummy values while keeping the structure. Since surrogate data is usually checked into source control or distributed to developers, treat it as you would treat any sample dataset – it should ideally be anonymized. Furthermore, if the application handles authentication or authorization flows, you'll need to decide how to handle that in surrogate mode. In many cases, teams bypass the auth in surrogate mode (since it's a trusted local context) or provide a dummy token that the app accepts. Care must be taken that any such shortcuts are only active in surrogate/offline mode and cannot leak into production code paths.

- **Complex API Logic or State:** Surrogate dependencies work best for request/response pairs that are largely static. If the real API has complex stateful interactions or time-dependent responses, those are harder to simulate with simple static files. For instance, consider a payment API that goes through multiple status changes – a static surrogate might only represent one snapshot of that process. Or an API that returns different data on each call (like an ever-incrementing counter or random value) – the surrogate would return the same data every time. In such cases, the surrogate may not perfectly mimic the live behavior. This is usually acceptable for front-end development (which often doesn't need to simulate every dynamic behavior, just the final outcome), but it's a limitation to note. If needed, one can enhance the surrogate mechanism with a bit of logic (for example, a small script to rotate through multiple JSON responses to simulate progression). However, that increases complexity and starts to approach writing a mini fake server. The general guidance is to keep surrogates simple; for truly complex cases, developers might still need a real environment or a specialized simulation.

- **Toggle Management and Risk:** Introducing a surrogate mode means there is extra logic in the code – logic that decides which data source to use. It's important to implement this in a clean, controlled way to avoid confusion. For example, one wouldn't want an application accidentally running in surrogate mode in production. Usually the toggle is wired to something very explicit (like a build-time flag, or a specific query parameter in development, etc.). Ensuring that this mode cannot be enabled in production builds (or if enabled, it doesn't have access to any stale data) is part of release management. Another concern is making sure developers don't forget which mode they are in. Clear logging or UI indicators (like displaying "[Offline Mode]" when surrogate mode is active) can help prevent situations where someone is hitting the surrogate and wondering why they don't see new data from the live server. Essentially, treat surrogate mode as a powerful tool, but handle it with the same rigor you would any feature toggle – with guardrails and clarity.

- **Perceived Reliability vs. Reality:** There is a subtle psychological effect to watch out for: if developers rely too heavily on surrogate mode, they might get a false sense of security that "everything works" because all tests pass and the app looks fine offline. Meanwhile, the real API

could be having issues or differences that are uncovered only later. To combat this, integrate real API testing into your workflow at sensible points. Surrogate mode should not completely replace integration testing with live systems, but complement it. For example, a good practice is to run a small set of smoke tests against the live API (maybe in a nightly build or before a release) using the same test cases that run in surrogate mode. This will catch any divergence. If both surrogate-mode tests and live-mode tests pass, you have high confidence. If surrogate tests pass but live tests fail, that indicates the surrogate data or expectations might be out of sync.

By being mindful of these challenges, teams can effectively use surrogate dependencies while mitigating risks. Many of these considerations (data syncing, security, toggling) can be addressed with automation and good discipline. In exchange for this effort, the payoffs – in productivity and software quality – are substantial.

Implementation Strategies

Implementing surrogate dependencies requires careful thought in the architecture of the application's front-end code, but it does **not** require heavy frameworks or external libraries. In fact, one of the goals is to achieve this with pure JavaScript/TypeScript, keeping things lightweight. Below, we outline key strategies and patterns for implementing surrogates cleanly:

Centralize and Abstract Network Communication

The foundation of a maintainable surrogate system is a centralized network layer. Instead of scattering `fetch()` calls (or Axios, etc.) throughout the codebase, the application should funnel all data retrieval through a small set of functions or a client class. For example, you might have an `ApiClient` object with methods like `getUser(id)`, `getOrders()` etc., or a generic helper like `fetchData(endpoint, params)`. By having this single chokepoint, you make it easy to insert the surrogate logic. All calls go through here, so this is where you implement:

```
const USE_SURROGATE = window.CONFIG?.surrogateMode ?? false;  // a global
flag set in config

async function fetchData(endpointPath) {
    if (USE_SURROGATE) {
        // Construct path to local JSON based on endpoint
        const surrogateUrl = `/surrogate_data/${endpointPath}.json`;
        const response = await fetch(surrogateUrl);
        return response.json();
    } else {
        const liveUrl = LIVE_API_BASE_URL + '/' + endpointPath;
        const response = await fetch(liveUrl, {/* credentials, headers, etc.
as needed */});
        return response.json();
    }
}
```

In the code above, `endpointPath` might be something like `"users/123"` or `"orders/list"` depending on how you structure it. The key point is that the logic to decide surrogate vs. live is **isolated** in one place. If tomorrow you rename the surrogate data folder or change how you fetch it, you update

it here only. Also, this makes it trivial to turn surrogate mode on or off via configuration. For example, you could have a build script that sets `window.CONFIG.surrogateMode = true` for a special "offline bundle," or toggle it via an environment variable in development. By avoiding direct network calls elsewhere, you also reduce the temptation for developers to accidentally bypass the surrogate (everyone goes through `fetchData`, no exceptions).

Another pattern is to use dependency injection or factory functions for the API client. For instance, you might have two implementations of an interface `DataProvider`: one that calls the network, one that reads local data. At startup, depending on mode, you supply one or the other to the app. This is more common in large applications or when using frameworks, but the idea is the same. The bottom line: architect your app such that *swapping out the data source is a one-liner*, not a refactoring nightmare. This also means avoiding deeply embedding URL strings for APIs in many components – keep them in a config or in the central client.

Organize Surrogate Data Mirrors

Your surrogate JSON files should be organized in a clear, predictable structure. Typically, the easiest approach is to mirror the API routes. For example, if your live API has endpoints like:

- `GET /api/users` – returns a list of users
- `GET /api/users/{id}` – returns details for a specific user
- `POST /api/users` – creates a new user (and returns the created record)
- `GET /api/orders?date=2023-01-01` – returns orders for a date filter

You can create a directory structure under a `surrogate_data` folder that corresponds to these. One scheme could be:

```
surrogate_data/
├── users
│   ├── list.json            ← data for GET /api/users
│   └── 42.json              ← data for GET /api/users/42
├── orders
│   ├── list.json            ← data for GET /api/orders (maybe general list)
│   └── date-2023-01-01.json ← data for GET /api/orders?date=2023-01-01
└── ... (more endpoints)
```

In this layout, we have a subfolder for each resource type, and within it, files for different queries or IDs. The naming convention should be intuitive. Here we used `"list.json"` for an index listing (no special query) and a combination like `"date-2023-01-01.json"` for a filtered query example. Some teams choose to incorporate the HTTP method in filenames (e.g., `GET_users_list.json` vs `POST_users_create.json`), or they create separate folders for GET/POST if needed. In most front-end uses, GET responses are the main concern, since POST/PUT requests in development are either less frequently tested offline or can reuse GET results for their outcomes.

It's often useful to include **metadata in the surrogate files**, especially if they were captured automatically. For example, a JSON file could include a comment (if using a format that supports it or a separate README) stating when and how it was obtained, or what endpoint and parameters it corresponds to. Keeping surrogate data well-documented helps avoid confusion. If multiple variants of

data exist (like multiple user examples), ensure each file clearly indicates its purpose (for instance, `user-42.json` vs `user-no-profile.json` for a user with missing profile picture scenario).

When serving these files, if you are running a dev server (like webpack dev server or similar), you just need to ensure the `surrogate_data` folder is served as static content. In a plain setup, placing it in the public directory would suffice. If using a CDN approach, you might upload these JSON files to a storage bucket and configure `fetchData` to pull from that URL (which might even allow non-developers to update test data by just replacing JSON on the CDN).

Switching Modes Safely

How the application switches between live and surrogate mode can vary. Some common approaches:

- **Build-Time Flag:** As mentioned, you might produce two builds of the application – one with surrogate mode enabled (for internal use) and one with it disabled (for production). For example, using a bundler's define plugin or environment variable: `SURROGATE_MODE=true npm run build` could bake in the surrogate endpoints. This ensures that no surrogate logic is present (or reachable) in the production build, eliminating any risk. The downside is maintaining two builds, but it's straightforward with modern CI systems.
- **Runtime Config:** Alternatively, the app can check a config at runtime. This could be a config file, an environment setting (for Node/Electron apps), or even a query parameter. For instance, the app could read `?surrogate=1` in the URL and turn on surrogate mode if it's present. This is handy for quickly toggling while testing ("just add `?surrogate=1` to use offline data"). For a more persistent setting, one could store the preference in local storage or have a debug menu in the app to switch modes. When using runtime toggles, make sure they are disabled or hidden in production, unless there's a valid reason to let end-users switch (usually not).
- **Detection of Offline State:** In some advanced use cases, the app might automatically fall back to surrogate mode if it detects the live API is unreachable (for example, if fetch requests are failing due to no network). This can be paired with a service worker that has cached the surrogate data for offline usage. This scenario overlaps with progressive web app (PWA) design for offline support. While not the primary focus of surrogate dependencies (which we frame mostly as a dev/test tool), it's a beneficial extension. Essentially, the surrogate data can double as an offline cache for end-users in a PWA, providing continuity when the internet is down.

Regardless of approach, it's a good practice to make the mode explicit. Logging which mode is active helps avoid confusion. Some teams color-code their UI (like a subtle background color change) or put an "[Offline]" badge when surrogate mode is on, purely to remind the user (developer) that "you're not hitting real servers now." This can prevent mistakes such as filing a bug that data isn't updating, when in fact you were looking at static surrogate data.

Example Scenario and Code Walkthrough

To cement these ideas, let's walk through a fictional scenario. Suppose we're building **"TaskHub,"** a project management dashboard (to ground it in a realistic application). TaskHub's front-end is a single-page web application, and its backend is a FastAPI service with endpoints like `/projects`, `/projects/{id}`, `/projects/{id}/tasks`, etc., which return JSON data. We want to enable a surrogate mode for TaskHub so that front-end developers can work on the UI even if the FastAPI service is not running.

Surrogate Data Setup: We identify key endpoints and use an integration test script to capture data. For example, we GET `/projects` to retrieve a list of projects, and save that JSON as `projects/`

`list.json`. We GET `/projects/alpha` (project with slug "alpha") to get a project detail, saving as `projects/alpha.json`. We also fetch sub-resources like `/projects/alpha/tasks` -> `projects/alpha-tasks.json`, and maybe a specific task `/projects/alpha/tasks/42` -> `projects/alpha-tasks-42.json`. For variety, we might capture another project as well, say `/projects/beta` and its tasks. We ensure to include an example of an empty list (maybe one project that has zero tasks, to see how the UI behaves). All these JSON files are placed under a `public/surrogate_data` folder in the front-end repository. The structure might look like:

```
surrogate_data/
└── projects
    ├── list.json
    ├── alpha.json
    ├── alpha-tasks.json
    ├── alpha-tasks-42.json
    ├── beta.json
    └── beta-tasks.json
```

(For brevity we omit some potential files; in practice you'd add what's needed.)

Application Code Changes: In the TaskHub front-end code, we likely have a module responsible for API calls. Before surrogate support, it might have looked like:

```
// apiClient.js (before)
const API_BASE = "https://api.taskhub.example.com";
export async function getProjectList() {
    const res = await fetch(`${API_BASE}/projects`);
    return res.json();
}
export async function getProjectDetails(projectId) {
    const res = await fetch(`${API_BASE}/projects/${projectId}`);
    return res.json();
}
// ... more similar functions for tasks, etc.
```

To add surrogate capability, we modify this module:

```
// apiClient.js (after adding surrogate mode)
const API_BASE = "https://api.taskhub.example.com";
// Assume a global flag or an import that tells us surrogate mode
const SURROGATE_MODE = window.SURROGATE_MODE === true;

async function fetchJson(url) {
    const res = await fetch(url);
    if (!res.ok) {
        throw new Error(`Request failed with ${res.status}`);
    }
    return res.json();
```

```
}

export async function getProjectList() {
    if (SURROGATE_MODE) {
        return fetchJson("/surrogate_data/projects/list.json");
    }
    return fetchJson(`${API_BASE}/projects`);
}
export async function getProjectDetails(projectId) {
    if (SURROGATE_MODE) {
        return fetchJson(`/surrogate_data/projects/${projectId}.json`);
    }
    return fetchJson(`${API_BASE}/projects/${projectId}`);
}
export async function getProjectTasks(projectId) {
    if (SURROGATE_MODE) {
        return fetchJson(`/surrogate_data/projects/${projectId}-tasks.json`);
    }
    return fetchJson(`${API_BASE}/projects/${projectId}/tasks`);
}
// ... etc for other endpoints
```

We introduced a helper `fetchJson` for brevity, and wrapped each API call with a conditional check. This is a straightforward approach. We could refactor it further to remove repetition (for instance, a generic getter that constructs paths), but clarity is often more important, especially when maintaining the surrogate files. Notice how the surrogate file paths are constructed – they match the structure we decided on. In surrogate mode, `getProjectDetails("alpha")` will fetch `/surrogate_data/projects/alpha.json` (which the dev server will serve as a static file), whereas in live mode it goes to the real API.

Running in Surrogate Mode: Let's say we want to start the app in surrogate mode for development. We can include a small snippet in our HTML (or set `window.SURROGATE_MODE` via a script) for when we want offline mode. Alternatively, if using a build flag approach, we might have a separate HTML or toggle to set that global. In our scenario, we'll assume the developer manually sets a flag in a config file for simplicity. Once that's done, they run `npm start` for the front-end, and because the dev server serves the `surrogate_data` folder, the app loads data from those files. The UI comes up showing, for example, two projects "Alpha" and "Beta" (from `list.json`). If they click "Alpha", the app calls `getProjectDetails("alpha")`, which returns data from `alpha.json`. It also calls `getProjectTasks("alpha")`, getting data from `alpha-tasks.json`. All this happens quickly and without errors, because the data and structure are exactly what the app expects from the real API.

If the developer instead started the backend server and turned off the surrogate flag, the *same code* would call the live endpoints and (ideally) get the same data. This dual-mode has been achieved with minimal intrusion in the codebase and no external libraries – just careful design.

Integrating with CI/CD and Team Workflows

To ensure surrogate dependencies remain effective, integrate them into your team's workflow: - **Automated Data Refresh:** If possible, have a CI job that periodically re-captures API responses (especially if the API is rapidly evolving). This could run tests against a staging deployment of the API

and push updates to the surrogate JSON files. Teams sometimes do this nightly or whenever a backend schema change is detected.
- **Verification Tests:** As mentioned, include a set of tests that run against both surrogate mode and live mode to flag discrepancies. For example, a test that compares the shape of data from `getProjectList()` in surrogate vs live to ensure no fields are missing in the surrogate JSON. This acts as a guardrail.
- **Pull Request Reviews:** Treat changes to surrogate data files like changes to code. If someone updates a JSON file (say to add a new field or scenario), the PR reviewer should consider why and perhaps double-check it matches the real API output. Surrogate data doesn't need the same rigor as code, but it's still part of the product.
- **Documentation:** Document how to use surrogate mode in the project README or developer guide. Newcomers should know what the surrogate_data folder is, how to regenerate data, and how to enable/disable offline mode. This avoids confusion where someone might think the surrogate files are unused or wonder why their local app isn't calling the API.

By following these implementation strategies and team practices, surrogate dependencies can be introduced in a robust, low-friction manner. The goal is to maximize the benefit (ease of use, productivity) while minimizing maintenance overhead and complexity.

Surrogate Dependencies in TDD and LLM-Based Development

Surrogate dependencies have a natural synergy with **test-driven development (TDD)** practices and are increasingly relevant in the context of **LLM-based development environments** (where AI assistants or agents are involved in coding and testing). Let's explore these connections:

Test-Driven Development: In TDD, developers write tests (often failing at first) and then implement code to make them pass, iterating rapidly. Surrogate dependencies can extend TDD from unit tests into the realm of integration tests. For example, a developer can write an integration test for the front-end: *"When I load the project dashboard, it should show 2 projects and their details."* Initially, without a backend, this test would fail because no data is coming in. But with surrogate approach, the developer can simultaneously create the surrogate JSON files representing the expected API responses (perhaps based on an API specification or using a dummy backend). They then run the app in surrogate mode to fulfill the test expectations. Essentially, the surrogate data serves as the expected outcome for the test – it is the "specification by example." Once the front-end code is implemented to parse and display that data, the test passes. This flips the usual need of having a live API available; instead the team can design API contracts and test against examples of those contracts even before the backend exists (or before it's accessible in the dev environment). It's similar to using mocks in testing, but much closer to real usage. Moreover, once the real API is available, the same tests can be executed against it to validate that reality matches the surrogate assumptions.

One can also incorporate surrogate data generation into TDD workflow. Consider that you have unit tests for the API (on the server side) that define what responses should be. Once those pass, you could automatically export those responses as JSON – which become the surrogate files for the front-end. Now the front-end can be developed and tested against exactly what the backend logic produced in tests. This creates a tighter feedback loop between front-end and back-end development. In scenarios where the backend is developed by another team or is an external service, surrogate dependencies allow your team to proceed with TDD for integration without waiting on the other side. You basically freeze a "contract" in the form of surrogate JSON and iterate on your side. Later, when integrating for real, any differences are quickly caught by tests (as discussed earlier).

LLM-Based Development Environments: The rise of large language model (LLM) assistants (like GitHub Copilot, ChatGPT, and others) in software development opens new possibilities and needs in the dev workflow. Surrogate dependencies can play an interesting role here:

- **AI-Assisted Coding:** When using an AI coding assistant, providing it with as much context as possible yields better results. If an AI is generating front-end code that calls an API, having example responses (the surrogate JSON) can help the AI understand the data structures. For instance, an AI that sees a sample of `projects/list.json` can infer that the project list is an array of objects with certain fields (id, name, etc.), and thus generate code accordingly. In an interactive environment, a developer could even share the content or schema of surrogate files with the AI to assist in prompt engineering. Surrogate data thus becomes part of the spec that the AI uses to align its output with the real-world use case.

- **AI Agents for Testing:** There is emerging experimentation with AI agents that autonomously test applications (for example, an agent that clicks through a web app and verifies behavior). In offline mode, such an AI agent can run the application without needing network access or credentials, thanks to surrogate dependencies. This sandboxed setup is ideal for an AI agent – it won't accidentally trigger real transactions or require a complex environment. The surrogate data provides a safe, consistent ground truth. An AI-based tester could even manipulate surrogate files to generate different scenarios and then reload the app to see how it copes. This is a powerful concept: using AI to fuzz or explore the state space of app behavior by varying the inputs (which are just JSON files). Because surrogate mode responses are deterministic and quick to load, the AI's feedback loop is faster, which is crucial for its effectiveness.

- **Natural Language Explanations:** If documentation is sparse, surrogate data can assist an LLM in answering questions about the system. For example, a developer might ask an LLM: "What fields does the project API provide, and how are they used in the UI?" If the LLM has access to the surrogate JSON and the front-end code, it can cross-reference to produce an answer. Surrogates essentially encode knowledge about the API that can be fed into an LLM context window. This is speculative, but as tools evolve, we might see IDE plugins where surrogate data is one of the sources an LLM looks at when generating or refactoring code.

- **Generative Testing and Data Synthesis:** Another angle is using LLMs to generate surrogate data. Suppose you have an API with many edge cases that are hard to produce from the real system. You could prompt an LLM: "Generate a JSON response for `/projects` where one project has a missing description, one has an extremely long title, and one has zero tasks." The LLM could output a plausible JSON matching the schema with those conditions. That JSON can become a surrogate file to test those edge cases in the UI. While caution is needed to ensure accuracy, this could dramatically reduce the manual effort in crafting test scenarios.

In all these ways, surrogate dependencies prove to be a forward-looking practice. They not only solve immediate problems (offline dev, testing) but also pave the way for better human-AI collaboration in coding. By having a self-contained environment where all external interactions are encapsulated in data, we create a playground where both humans and AI can safely and efficiently co-create software.

Best Practices and Tips

To conclude the technical guidelines, here is a summary of best practices for implementing and using surrogate dependencies effectively:

- **Plan Early:** If you anticipate the need for offline mode or easier testing, design the API integration with surrogates in mind from day one. It's easier to build the abstraction upfront than to retrofit dozens of API calls later. Early planning ensures you avoid anti-patterns that complicate surrogate introduction (like tightly coupling UI components to fetch calls).

- **Keep Surrogate Mode Logic Minimal:** The surrogate toggle should be a simple boolean check that swaps URLs or data sources. Avoid spreading complex conditional logic throughout the code ("if surrogate do X else do Y" in many places). Ideally, beyond the network layer, the rest of the app shouldn't care whether data came from live or surrogate. If you find yourself needing special-case code deeper in the app for surrogate mode, reconsider your approach – perhaps you need to refine the surrogate data to better mimic live conditions.

- **Regularly Update and Validate Data:** Treat surrogate data as versioned artifacts. Update them when APIs change (don't let them stagnate for months while the real API evolves). When you do update, run a diff on the JSON to see what changed – this can alert developers to new fields or removed fields, which they might need to handle in the UI. Use automated tests to validate that surrogate JSON adheres to expected schemas (you could use JSON schema validation in tests if schemas exist).

- **Isolate Sensitive Info:** Ensure no secrets (API keys, user passwords, etc.) are stored in surrogate files. If your capturing method automatically dumps headers or tokens, strip those out. A good practice is to have the capture script sanitize the output (e.g., replace user emails with "user@example.com"). This way even if the surrogate data is public (or accessible to all developers), it's not a security risk.

- **Leverage Surrogate for Onboarding:** When a new developer joins, have them run the app in surrogate mode first to verify their setup. It's a quick smoke test – if the app doesn't run with surrogates, something's wrong in the front-end environment (which is easier to troubleshoot without also involving backend issues). Once that works, they can switch to live mode if needed. This reduces the cognitive load initially (they can explore the UI with known data, as opposed to possibly empty or erroring screens if they had no data).

- **Avoid Surrogate Creep into Production:** Double-check your deployment pipeline so that surrogate files aren't accidentally deployed to production web servers (unless intended for offline PWA use) and that any surrogate flags default to off in prod. Some teams even put a runtime assertion: if `SURROGATE_MODE` is true but the app is not running on localhost (or not a dev build), then throw an error – as a safety net.

- **Document Known Limitations:** If your surrogate doesn't cover certain features (for example, maybe real-time updates or websockets aren't easily simulated, so those just don't function in offline mode), document that for users of surrogate mode. It's fine if not 100% of functionality is offline – usually, it's 90% that matters, and the rest can be tested with the backend as needed. But letting developers know "Feature X requires live mode" saves time when they hit it.

- **Community and Sharing:** If multiple projects or teams in your organization use similar APIs, consider sharing surrogate data or standardizing the approach. For example, if there's a common user profile service, one team's surrogate JSON for `/profile` could be reused by another team's app that also calls `/profile`. This can foster collaboration and ensure everyone benefits from improvements. In open source contexts, surrogate data (if generic enough) could even be shared as sample data for others to run your project.

- **Think of Surrogates as Tests:** Finally, view the surrogate dataset as a large, living integration test. Each JSON file is effectively asserting "the system returns this structure for this request." In fact, some organizations keep surrogate responses alongside expected outputs in test cases (especially for back-end contract tests). This mentality helps maintain diligence – when an API contract changes, you update the surrogate (the test data) and any affected front-end logic together, ensuring consistency.

By adhering to these best practices, teams can ensure that surrogate dependencies remain a help and not a hindrance. When done right, maintaining the surrogate mode becomes a normal part of development – a small upfront cost for a significant payoff.

Conclusion

Surrogate dependencies offer a powerful paradigm for modern software development: they empower developers to simulate and **stand-in for complex backends with simple JSON files and toggled logic**, unlocking offline development, faster testing, and greater resilience to change. We have discussed how surrogate dependencies work, how to implement them in a clean and maintainable way, and the many benefits they bring in productivity and software quality. By comparing them with traditional mocks, stubs, and service virtualization, we see that surrogates hit a sweet spot of realism and simplicity – they leverage real data to avoid the brittleness of hand-written mocks, yet remain straightforward to set up compared to full-blown service virtualization platforms.

This white paper, co-authored by Dinis Cruz (who has applied these concepts in numerous projects) and ChatGPT Deep Research, synthesizes both practical insights and forward-looking implications. Surrogate dependencies are not just a theoretical idea; they have been used in the field to great effect – for example, allowing apps to run entirely offline during security testing engagements, or enabling rapid prototyping of front-ends against evolving APIs. As Dinis Cruz highlighted in his OWASP presentation, such approaches help align what developers want (fast, flexible workflows) with what AppSec and QA teams want (reproducible, testable systems) [1] . They essentially create a common layer – the surrogate data – that all stakeholders can refer to.

Looking ahead, the importance of surrogate dependencies is likely to grow. With more developers working remotely and asynchronously, having a self-contained development environment is invaluable. With microservices and third-party APIs proliferating, being able to decouple your development from external changes provides stability. And with AI increasingly in the mix, providing a controlled sandbox (rich with real examples) will be key for AI-assisted development and testing. Surrogate dependencies address all these needs.

In conclusion, adopting surrogate dependencies is a design decision that pays off by making your project more robust to external factors and more amenable to rapid, iterative improvement. It is a technique that encourages better understanding of one's own system (by explicitly capturing its inputs/outputs) and yields a safety net for development ventures large and small. We encourage teams to consider this approach, start small by capturing a few key endpoints, and experience the difference it

can make. As the old adage goes, *"Trust, but verify"* – surrogate dependencies let you do exactly that with your integrations: trust that your app works offline with known good data, and verify against the real systems with confidence once you go online [11] [9]. It is our hope that this paper has provided both the conceptual foundation and the practical guidance needed to leverage surrogate dependencies in your own projects, thereby fostering a more efficient and resilient development process.

Sources: The concept and practices described here draw from industry experience and prior art, including Dinis Cruz's presentation on Surrogate Dependencies [6] [2] and writings on DevSecOps practices [12] [13]. These sources underline the critical role of offline-capable development and using real data for testing. Developers are encouraged to refer to those materials for a deeper understanding and for inspiration on extending surrogate dependency patterns to their specific tech stacks.

---

[1] [2] [6] [8] Surrogate Dependecies (in NodeJS)- v1.0
https://owasp.org/www-chapter-london/assets/slides/OWASP20160929_NodeJS_Surrogate_Dependencies.pdf

[3] [4] [5] [7] [9] [10] [11] [12] [13] Read SecDevOps Risk Workflow | Leanpub
https://leanpub.com/secdevops/read