accusoft®

How PrizmDoc Editor Stays RESTful

Emily Kaneff
Software Engineer II, PrizmDoc Editor

In the modern web, the concept of being “RESTful” is commonplace. The design concept has been around for many years now, which means there are a million and a half existing articles and tutorials that explain what that person believes are the fundamentals, so I am going to try and avoid doing that here. Instead, I want to give a light overview of the generally agreed upon pillars of what makes a concrete RESTful service and how we have been able to apply those pillars to our own PrizmDoc Suite plus any specifics that we have decided were best suited for our project’s needs.

To begin, just in case you are unfamiliar, REST is a set of principles that one can choose to use to design a web service project. It focuses on managing a system’s resources and how to handle resource and application state over HTTP. Across the many articles that talk about the fundamentals or best practices of a RESTful web service, there seem to be a few consistent themes that we can extrapolate and identify as the basic principles that we will use as a reference in this article:

- Explicit use of HTTP methods (verbs)
- Statelessness
- Using JSON (or XML) for data transfer
- Proper use of status codes

Explicit Use of HTTP Methods

For the sake of consistency and readability, a RESTful web service should use HTTP methods (GET, POST, PUT, DELETE) as they are defined in their protocol.

In PrizmDoc Editor, we have a collection of documented public API routes that an integrator will need to use to manage their documents and sessions. For documents, we allow for both uploading and downloading a document:

Download Document

```
GET /api/v1/documents/:documentId
```

Upload Document

```
POST /api/v1/documents
```

Since both of these routes go through the same service (using the same API base route), it is important that we are explicit with which HTTP method we use in order to avoid confusion for both the integrator as well as our own development team. For all of our GET routes, as defined in the HTTP method protocol, it is expected that no changes will be made to any data by the server, but rather it will fetch the requested resource and return it with no side effects, whereas POST will, in fact, trigger an update to the internal document store with the addition of a new document.

In addition to the correct use of the methods, we are also utilizing another important REST principle in how we structure our URIs. It is generally agreed upon that within a RESTful web service, URIs should be structured in a way that makes it predictable and easily understood as to what that route is going to do. We want to avoid using query strings as much as possible, and we also don't want to have route names that state what their operation is (ex: /getDocument), so we combine the practice of explicit HTTP methods with a URI naming convention that uses a directory-like structure (starting with a base and getting more specific as you go) in order to make our API as clear as possible:

```
GET /api/v1/sessions/:sessionId/document

GET /api/v1/sessions/:sessionId/document/:format

GET /api/v1/sessions/:sessionId/document/files

GET /api/v1/sessions/:sessionId/document/files/*
```

These routes also demonstrate another RESTful practice we utilize here, and that is versioning. Part of our base API path includes a version number (v1). This makes it easy to upgrade the API when breaking changes are introduced so that we don't break existing users with new functionality.

Statelessness

In order to ensure that the service scales and works well within server clusters, REST recommends that requests contain all the information they need to generate a response within itself (in the headers and the body) so that there is no state held locally on the server between requests.

For example, PrizmDoc Editor has a public POST route that can be used to create a session. This route takes in body data that is used to configure the session, such as a document ID, user data, and optional configuration overrides:

```
request.post({
  headers: {
    'Content-Type': 'application/json',
  },
  json: {
    documentId: 'b43f1013-158d-4bd6-ab30-b2ac19a5cf97',
    user: {
      uniqueId: 'some-test-user',
      displayName: 'Test User',
      initials: 'TU'
    }
  },
  uri: 'http://localhost:21412/api/v1/sessions'
}, (err, response, body) => {
  if (err) {
    return console.error(err);
  }
  console.log('Session successfully created');
  console.log(body);
});
```

The data provided in the body of the request is all the session service needs in order to generate and respond with a session ID, making it super easy for this request to land on any server after being load balanced and guaranteed a consistent response.

Using JSON (or XML) for Data Transfer

When choosing a format for the data that your RESTful web service is going to be exchanging, it's important to keep it simple and easy to read so that both the integrators and internal developers are able to keep track of those resources smoothly. Which format to use is mostly based on your application's use case, but whatever you choose, you want to make sure you are consistent.

In the previous section, there was an example of what a POST request to the session service would look like if you were trying to create a session. In the header of that request, we specify a content-type of 'application/json' for the body of the request, since we have decided to use JSON format for our services. This also means that we will always return JSON in the body of the response as well when necessary. For example, when getting the default session configuration that is used by all sessions:

```
request.get({
  json: true,
  uri: 'http://localhost:21412/api/v1/sessions/configurations/default'
}, (err, response, body) => {
  if (err) {
    return console.error(err);
  }
  console.log(body);
});
```

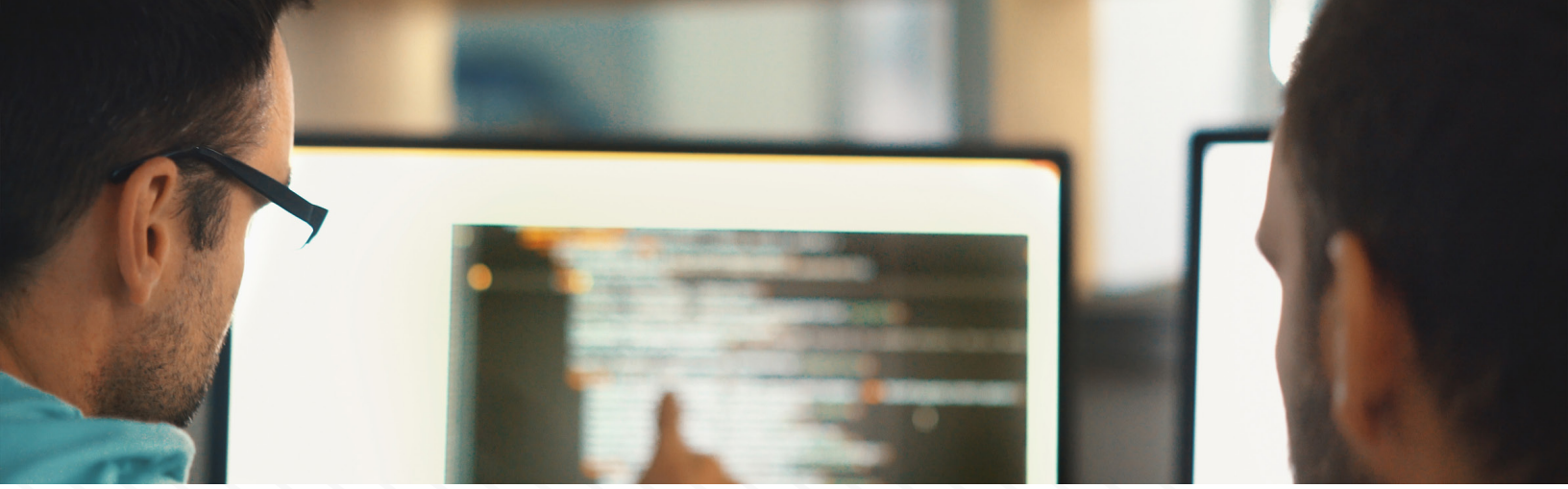
That route is designed to return a response body that will be in JSON format:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  “branding”: {
    “visible”: false
  },
  “menubar”: {
    “items”: [
      {
        “name”: “File”,
        “items”: [
          “New”,
          “Open”,
          “-”,
          “Rename”,
          “-”,
          “SaveDownload”,
          “-”,
          “Close”
        ]
      }
    ],
  }
  “toolbar”: {
    “items”: [
      “Bold”, “Italic”, “Underline”, “-”, “LeftAlign”, “RightAlign”
    ]
  },

“statusbar”: {
    “items”: {
      “left”: [
        “Bold”, “Italic”, “Underline”
      ],
      “right”: [
        “EditMode”
      ]
    }
  }
}
```

Using JSON is something we decided to do as a company across our PrizmDoc Suite of products, but you can choose the format that best suits your project and your needs.



Proper Status Codes

When a client makes a request to a server through an API, the server should be communicating clearly back to the client with useful and accurate status codes. Using every single one of them is a bit excessive, but having an array of some of the more commonly known and understood ones will go a long way in your error handling. Some of these may include:

- 200 OK
- 201 Created
- 204 No Content
- 400 Bad Request
- 403 Forbidden
- 404 Not Found

For HTTP 500-level status codes, that typically implies that something went wrong internal to your service. The problem can be unclear, so how you want your client and server to react to those errors is up to you.

At Accusoft, within the PrizmDoc Suite, we opt to never return a 500 error to the client, but rather apply the appropriate 500-level status code if possible, and default to HTTP 580 with a specific errorCode value in the JSON body of the response:

```
HTTP 580 InternalError
Content-Type: application/json

{
  “errorCode”: “InternalError”
}
```

Using status codes properly can make it easier to test your product as well, since you will be able to make assertions on specific codes and messages to ensure you know that your product fails or passes in the exact way it is meant to.

Conclusion

At the end of the day, REST is designed to be visible, readable, and simple. Its pieces should be self-descriptive and independent to allow for easy scaling. Within the PrizmDoc Suite, we do our best to stay as close to the REST principles as we can. When designing your own application, it's important to be clear on what your own project goals are before deciding if REST suits you and your needs, since it may not be a perfect fit. You may even need to tack on your own custom rules in order to accomplish everything your application needs, similar to how we have custom error code customizations for 500-level errors in some cases. Above all, you just need to be consistent in your design and document everything along the way.