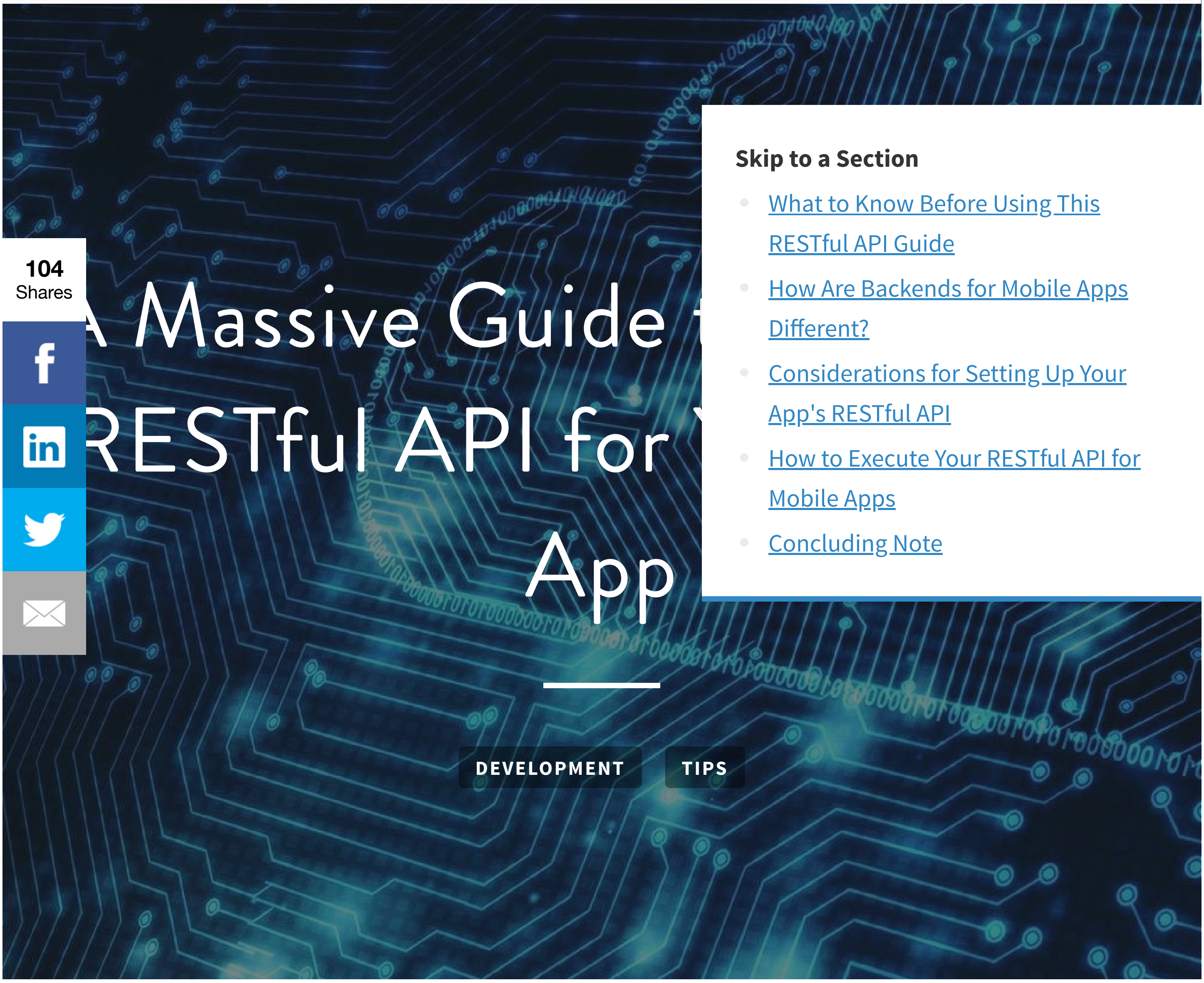


104
Shares

A Massive Guide t
RESTful API for Y
App

DEVELOPMENT TIPS

Skip to a Section

We build apps of all shapes and sizes here at Savvy Apps, but a common element is that they communicate with servers. Very few apps today operate without some sort of Internet connectivity, meaning that they interact with a backend, web services, or APIs. These APIs could be provided by Google, Amazon, Facebook, or comparable third-parties. They also could be APIs that are developed internally.

The problem for these internal or in-house APIs are twofold. Many don't take the time to plan out a *good* API. Additionally, even with the abundance of apps, not everyone has built or APIs for apps specifically. In our experience, we've found that establishing guidelines on how to build better APIs for mobile apps saves time and effort during development and reduces headache later on in the process.

We wrote this guide to outline the best practices for bu databases for mobile apps and mobile clients. In this p RESTful API specifically for mobile apps. This informat It's helpful for our customers and any other web or bac to properly build and maintain their own app-focused

TL;DR

- Use a well-known architecture so new develop
- Make the server do the heavy lifting so mobile c
- Version your API so it handles requests coming
- Account for offline usage and usage across devices.
- Prioritize performance and scalability when picking where to host your server.
- Use standard security protocols and well-vetted authentication/encryption libraries.
- Build three backend environments: development, staging, and production.
- Let your data decide the type of database you use.
- Construct API URL endpoints so that it's very clear what that resource contains.
- For requests, let the client send full objects, and the server use the fields it needs.
- Utilize UTC for dates/times, and let the client figure out how to display the data.
- Remember that GET and PUT requests need to be idempotent.

What to Know Before Using This RESTful API Guide

REST is by far the most commonly-used style for designing APIs, especially in the mobile world. There are also particular subsets of REST, like OData, that further define how data should be transmitted between your apps and the server. While those subsets may be best for your particular needs, we're going to keep the conversation broad enough to cover all REST styles. Adhering to a popular, generic, RESTful architecture style will ensure that new developers tasked with maintaining your server code in the future will be familiar with how it works and, more importantly, how new services should be built onto it.

we'll also be discussing RESTful APIs through the lens of mobile app systems. These rules, however, will certainly help with supporting web apps and other systems with your API too. In most cases, the mobile app client asking for resources and the backend server handing out those resources are going to be written in different programming languages and often by different development teams.
out below will ensure that both teams are setting and
when the time comes for the two platforms to commu
Worse than thinking that you've completed a new end
format the mobile client can use efficiently. In projects
members, it is critical for everyone to stay in constant
shared blueprints to avoid unexpected miscommunica
Adhering to a commonly-agreed set of standards and
iterate faster and more efficiently, which makes develo
expensive in the long run.

Skip to a Section

How Are Backends for Mobile Apps Different?

Before we dive into the hosting, security, architecture, and other considerations for creating your RESTful API, let's examine what makes building an API for mobile apps different from other systems. These mobile-specific concerns are essential to making sure your RESTful API is prepared to work efficiently with a mobile app and the expectations of its users.

HTTPS, Not HTTP

The internet was built on HTTP, but mobile platforms enforce HTTPS requirements with modern encryption and trusted signed certificates. A mobile backend needs to use HTTPS for every endpoint. Your development, staging, and production environment servers should all be using the same type of signed certificates. This will save you headaches later when migrating/testing features on each environment, allowing you catch security issues upstream before they become a problem on the live server and start affecting real users.

Server Does Most of the Work

To save on network data costs and battery life for users, you typically want mobile clients doing as little work as possible. It's quite rare to see a mobile app that couldn't benefit from storing data remotely or offloading burdensome and sensitive tasks to more powerful, remote machines. Plugging into a remote API keeps your stored data secure and allows your app to continue running smoothly for the user, staying focused on presenting your data quickly instead of calculating. Mobile app users expect their data to be synced across all their devices, which is also solved by moving

Skip to a Section

Letting the server do the heavy lifting also saves time f platforms by moving and consolidating code to your s both your iOS and Android apps. The server is going to hardware than the mobile clients for number crunchin

Server Issues Can Kill an App

You can also expect any errors a mobile user experienc megaphone. Tech users these days have little patience when something doesn't work as expected, especially mobile app users. There's no bigger stage for issues to be shared than in App Store and Google Play app reviews. If something goes wrong, the server needs to respond with user-friendly error messages or error codes the client can use to assuage the user and, hopefully, help fix the issue. Even a single error can cause a 1-star review and positive reviews are critically linked to the success of an app. Too many negative reviews caused by server issues will stop new downloads for your app.

Versioning is More Important

With mobile app users updating their apps (or not) at different frequencies, versioning your API becomes more important than other, more controlled environments. With several different versions of the app running in the wild, the server needs to consolidate and handle the various requests coming in from new and legacy users alike. We'll dig into effective strategies on how to handle this later.

Plan for Push Notifications

A useful communication avenue unique to mobile is the push notification. There are third party tools that specialize in push notification, but sometimes you need to manage the process yourself. Your server may be responsible for tracking device tokens that maps devices to users for sending push notifications. Using a service like Firebase, however, to e tokens and send push notifications to iOS and Android users is typically more cost effective than building your own.

Reconciling Offline Activity

Many mobile users will expect the app to have some li
Once reconnected to the server, reconciling the offline
needs to be considered. This is especially important fo
multiple devices, such as their phone and tablet. Coor
and order of operations is something that needs to be
backend developers.

Skip to a Section

Considerations for Setting Up Your

Now that we've looked at what sets mobile apart from other systems, we can dive deeper into planning your RESTful API. These tips address common concerns for hosting the server, dealing with security, creating the backend architecture, choosing database and storage options, using the right tools, and supporting multiple platforms.

Hosting the Server

Choosing the location to host your server is a big decision. If you don't have the desire or capacity to host your own bare-metal server, there are plenty of cloud-hosted solutions available these days. Every project is different, each with specific needs for performance, scalability, and administrative features. Some core factors to pay attention to while evaluating services for where to host your server include:

- How does the service scale its resources? This could be horizontally (adds more machines) or vertically (upgrading hardware resources for existing machines).
- How will the cost increase with usage?
- Are there any migration features that would allow you to easily stand up multiple environments (development/staging/production) for your project?
- What features are already baked into the system that would save you time and energy from trying to recreate?

steps does the service take to protect your system (i.e. encryption, automated backups, uptime monitoring)?

- How easy can the data be ported to another service or platform?

Protecting the Data

104
Shares

Depending on your needs, you have a wide array of au
Any hosted service you choose should already include
trusted CA certificates. HTTP Basic Authentication is th
the least secure. OAuth2 is widely accepted as a secure
authentication and is highly recommended. There are
phone number authentication you could use as well. D
authentication! There is no need to reinvent the wheel
protocols and libraries that have already been vetted b
server side. Protecting each API endpoint behind authentication requirements should be the norm. Don't allow free passes on a resource unless necessary for functionality.

Skip to a Section

Sensitive data should be protected. This is a given, but security is a spectrum, not an absolute. Encrypt your user's sensitive data. Encryption may not be necessary for every project, but it should always be considered. Don't store your passwords in plain text. Please. Not only should you hash passwords, but using random salts for each password will significantly improve security.

Planning the Architecture

As we already discussed, you're hopefully planning on building not one, but three discrete backend environments: development, staging, and production. The development environment is where frequent development changes are rolled out as they're completed by developers. Data here can be generated by developers; this can be achieved through automated scripts to populate a database with a healthy amount of test data. As code passes through all its tests in continuous integration (hopefully you're testing your server's logic and API endpoints) and gets the OK from QA, it graduates into the staging environment.

The staging environment is going to try to resemble production as much as possible. Ideally, data here is an import of transformed real/live sampled data stripped of personal information. The more realistic data used here, the better confidence you'll have of how your system will perform in production. Porting data may not be feasible for your project, but having some sort of quasi-representative data in t
reducing risk and discovering bugs in the logic before
cannot be reproduced, having at least roughly the sam
production in these environments again will pinpoint
Otherwise, irreproducible slow downs, hang ups, and
can be a nightmare to track down in development.

Making Database and Storage Decisions

No matter what type of database you use, it's worth no
randomly generated UUIDs, not sequential. This helps secure resources by making IDs much harder to guess. When it comes to storing your data, you might be considering a traditional relational database like MySQL or MariaDB. Or maybe you prefer the scalability of a noSQL document database like MongoDB. Or perhaps you prefer the flexibility of a hybrid approach that something like PostgreSQL can offer with both relational or document storage support. Which database your project should use is really going to depend on your data. Here are some notes to cover the basics:

MySQL/MariaDB

- Well established, stable and reliable
- Lots of libraries, frameworks and tools to leverage
- Data is rigid, well structured and defined, leading to less data mismatch bugs
- Scaling requires a lot of domain-specific knowledge

MongoDB

- No tables, no formal schema, non-relational
- Easier to scale than SQL-based databases
- Easy ramp up and iteration of database model
- Easier to shoot yourself in the foot

PostgreSQL

- id and popularity is growing rapidly
- Built around giving more features and tools for the admins
- Flexible enough to mix relational data with model-independent data

Cloud document storage

- A cheap solution could be something like using sets of whole documents

Finding the Right Tools

You're going to need the right tools to get the job done between project teams, current team members, or futu onboarding process for your project, communication i project. During development, make sure you're using a teammates have access. We're fans of Pivotal Tracker and Trello, but any similar tool will work.

The point is to keep progress out in the open and maintain a historical record. This is especially important with multiple developers on a project. The developer responsible for writing the code to log in a user in an iOS app will really want to know when the server-side authentication API is ready to consume. They may even have a discussion around shifting priorities of the server developers to complete that task sooner rather than later if it's blocking the iOS app from progressing further. Letting the whole team have knowledge of not only what's currently being worked on but also what's up next will allow a more fluid workflow of coordinating work schedules to complete tasks in the most efficient way possible. This will also help circumvent issues where the API may have changed prematurely, causing a delay or otherwise negatively impacting development in the mobile app. It's also worth noting that just like any software iteration, release notes are invaluable when a new API is deployed, even during development.

When done right, RESTful API endpoints should be easy to test and should have tests covering both obvious use cases as well as expected edge cases for each endpoint. One of the core principles of REST is stateless, which makes our API endpoints small, modular black boxes, ripe for testing. New data comes in, successful message comes out, and the newly persisted data changes are easily verifiable in the database. Request for data goes in,

…which response comes out. The testing framework to use will of course depend on the language you're writing the backend server logic in. No matter how you write your tests, they should be well-maintained and run with 100% passing rate before every deploy.

During development, your testing suite may prove suf developers to review how the system works. Eventuall english, how the system you built actually works. I knc most of us, but it is critical for the success of a project is only as good as the effort you put into it. Fortunately tools out there that can do a lot of the work for you. Or tools is Postman . Not only is it useful for exploring or generate all the requests, responses, and handled errc reference later. You can literally build your documenta suite for your new API.

104
Shares

No matter how you document your API, try to include successful response codes, sample requests, and examples of both successful and failed responses, with expected error codes and messages. You also need to make sure this documentation is accessible to everyone on the team. It does no good if no one else can read it! Included below are some other documentation tools you may want to look at:

Swagger

- Can support bottom up or top down (contracts first, then write code)
- Language neutral, but does support a large variety of languages via plugins

Apiary

- Built on top of open sourced API Blueprint

Slate

- Supports Markdown
- Can be hosted on Github

Supporting Multiple Platforms

If you're building an API for a mobile app to consume, chances are good that you'll need to support other platforms in the future. The general rule of thumb for building APIs for mobile is to make the client as dumb and thin as possible, while keeping all the heavy sorting, filtering, number crunching, data aggregating, and consolidation on the server. This leverages the more powerful hardware of the server ar while fetching and showing the data to the user as qui

104
Shares

This is important because when you're building your a don't want to rewrite complicated filtering and parsing For this reason, you'll want to allow robust sorting anc fetch only what it needs. For example, don't return eve client app to sort through the data to find what it was lists of data to avoid overwhelming both the client anc determine how many results it should get back. While be beneficial to capture device names, OS versions, and types in request headers since those can be useful while reading logs and debugging in the future.

How to Execute Your RESTful API for Mobile Apps

Now that we've discussed how and where you should set up your server, let's dig into how to actually design your API in a RESTful manner. This section runs through how to use proper url paths, requests and responses, and other guidelines for implementing your RESTful API.

Handling URL Paths

For the rest of this guide, we're going to assume we're building an API for a library. For example, some of your resources might be a book, a book club, an author, a user, or a library location. The client would want access to all of those resources via the API so you could construct some endpoints mapping to:

```
GET    /books
POST   /books
GET    /books/{bookId}
PUT    /books/{bookId}
PATCH  /books/{bookId}
DELETE /books/{bookId}
```

```
GET       /books/{bookId}/authors
          /book
GET       /users/{userId}
GET       /users/{userId}/favoriteBooks
```

Notice the pattern emerging. We hope to construct our API URL endpoints in a way that makes it very clear what that resource contains. Now l design tools: nouns and verbs. The actions GET, POST, meaning they are the HTTP methods telling the server path itself tells you the nouns, which are the resources to make this distinction here, because otherwise you c difficult to follow.

Skip to a Section

Now take a look at these (bad) endpoints:

```
/getBooks
/createNewBook
/checkOutBook/{bookId}
/returnBook/{bookId}
/addBookToFavorites
/addNewMemberToBookClub
/changeBookClubMeetingTime
/changeBookClubMeetingLocation
/removeBookClubMember
```

You may glean more about what those API endpoints might be trying to do at a glance, but now you're going to have to add a new API endpoint for every possible action on that resource. Doing so will lead to frustration for the clients trying to consume your API. You want to keep your API concise and modular. You're not only creating an API for your apps right now, but you also need to think about how additional features or data types will fit into the API.

To improve your API's robustness, just let the requests drive the server's response. Now this may seem like a silly statement, because of course a client will make a request and the server's job is to send a response. If the server is dictating what, how, and where a client can fetch or change data, then that means that every time they want to perform a new action or feature, the client developers are going to have to wait for changes to be made to the server. The server is allowed to refuse requests. That's what error messages are for. But by building

Now, just because we're using nouns to drive the URLs, doesn't mean our resources need to be the same as our data model objects. The URL shoul appropriate, but we want to make these paths easy to resource `/favoriteBooks` probably will just return provide the clarification of what type of books we're g allows you move the logic of finding, filtering, and calc server, again allowing the mobile client to remain thin data and presenting it to the user.

This leads directly into how to handle filtering, sorting four sub-actions are all related as they let the client de retrieve from the server. Again, we want to construct our APIs to give the clients as much freedom as possible here, without having to exert too much extra effort on our side. We handle filtering, sorting, pagination, and searching with URL parameters. This allows us to keep our API endpoint quantity low and manageable while giving the clients the tools they need to perform more complicated actions. Once we add the ability to handle these types of query parameters, we open up the door for the client to make changes to better serve up content to the users as they see fit. Today, maybe they want to show all libraries in alphabetical order, but tomorrow they realize that sorting them by geographical location is more useful to end users. Or maybe they want to let end users choose how they want to see libraries sorted as a configurable setting. The client can do all of that without any extra work on the server side, which is what we want.

Here are some examples of handling these URL parameters for our library API:

```
GET    /libraries?sortBy=name&isCurrentlyOpen=true&pageCount=10
GET    /books?queryTitle=Sherlock+Holmes&queryAuthor=Arthur+Conan+Doyle
GET    /bookClubs?genre=mystery
```

API versioning is another feature we should implement to achieve the robustness that is especially important for mobile apps. Mobile developers don't always have the luxury of forcing software updates for all end users, so our API is going to have to be able to handle build new requests. We'll do this by routing requests with a version number. Some will argue that this version number should go in the URL path, and some will argue it should be placed in the request header. We prefer it to be in the URL path for easier discovery, but that's up to you and your team to discuss. We treat versioned resources as different resources, and that's why we think they deserve a unic value. Either way, the benefit of versioning API is allow while not interfering with legacy requests.

```
GET    /v1/books
GET    /v2/books
```

Another important guideline to call out with all of our from left to right, from broad to specific. You're letting endpoints, starting with the root single point of entry, letting related resources cascade from there. Again, the aim is to make these APIs easily explorable and intuitive for the clients to navigate. Determining what resources to actually expose in your API is going to be up to you and your dev team to decide. If you have any UI or UX designs for how the client apps are going to work, it's important for the developers working on the backend to be aware of what data they clients will need and when. Ask yourself questions like:

- Does a user need to log in to see this resource, or is it public?
- Does a user need to choose a library or city before viewing book clubs, or are they not attached to a particular location?
- What actions does the UI allow when viewing the details of a particular book, and how can we design the API to allow the client to best perform those actions?”

Rules for Requests and Responses

So we've laid out how we'll construct our API endpoints, now let's talk about how to communicate effectively within each of those endpoints. Let's start with the request. When handling a request, don't force the client to only send one or two fields. Allow them to send full objects if they wish, while the server just uses the fields it needs. If a request does come in with only a few fields, don't assume the missing fields are `null`. Fields that are `null` should be stated explicitly in both requests and responses. This prevents the other end to guess if the data really is null or just not included. Take a look at the example below.

```
PATCH /books/123
```

```
{
    "author": "Arthur Conan Doyle"
    "publicationYear":  "1902"
}
```

While this is not a complete book object, we'll build th
important information. Here the client had made it cle
id 123, as told by the URL. The request body tells us th
publication year. Only these fields should be updated
other details are not important to this request, and the
mean the title of the book should be set to `null`, it should just remain untouched. Allowing these partial object requests reduces parsing and handling of extraneous duplicate data.

Pay attention to the `Content-Type` and `Accept` headers in the request. They'll likely remain `application/json` for most requests, but if you need to support xml, this is the place to do so (not in the URL!). Other types of resources like files, images, audio, etc. should have their content type set and respected here as well.

After handling the request, we need to send a response. Things might not have gone the way the client expected and an error will need to be returned. Or maybe the request was processed successfully and simply doesn't require any data returned back. Or maybe everything went exactly as expected. This is where the response HTTP status code helps us out.

Be deliberate with what status code is returned, with 200, 201, and 204 for successful responses, and 4XX codes for errors. You can bet the clients will be paying attention to these codes, so we need to be careful with what we send back with our response. In every response though, we need to remain consistent. We should not be mixing "camelCase" and "snake_case" for our JSON keys. The norm is camelCase, but an argument could be made for snake_case being more legible. Whatever you choose, use it everywhere. Also, be about a book's ISBN. Pick one and use that key everywhere you need to pass that value.

When sending responses that could have one or many
object versus an array. If the client is requesting a colle
put it in an array and return it that way all the time. Ha
different for every programming language, but if you ju
trying to parse special cases on any platform.

The flip side of this advice is don't wrap your response
unless it provides value. Wrapping an object in a “dat
and time. As a follow up, don't include metadata on th
Ideally, we want the data model of the server and clier
throwing in extra fields that aren't part of the requested object will just get in the way. Below are some examples of what **not** to put in your responses:

Don't wrap data with useless envelopes.

```
GET /authors?genre=mystery

{
    "data": {
         [{ "authorName": "Arthur Conan Doyle",
         ...
         }]
    }
}
```

Don't return a single object when the client should expect a collection.

```
GET /authors?genre=mystery

{
    "authorName": "Arthur Conan Doyle",
    ...
}
```

Don't include metadata that isn't relevant to the client requested data.

```
    {
        "requestInfo": {
            "genre": "mystery"
        },
        [{
            "authorName": "Arthur Conan Doyle",
            ...
        }]
    }
```

When dealing with dates and times, remember that co
dates in ISO 8601 format with UTC values. Don't let the
display format, or determine what precision the date-t
apps will figure out how best to display the date and ti

Skip to a Section

Aside from reading a request and creating or finding it
going to need some programmed logic to best execute each request. We need to keep in mind when writing this logic that GET and PUT requests need to be idempotent. This means that no matter how many times a client might call `GET /books` on our server, the data will remain unchanged. We should avoid doing anything “extra” or behind the scenes that a client might not expect. Whatever logic or indirect actions the server needs to execute in response to a request needs to be communicated and documented to avoid surprises later.

Concluding Note

Hopefully, this guide has been insightful. When designing and creating our own backends, we focus on placing resources in a modular, explorable, and extendible way and communicating our decisions and priorities effectively. Constant collaboration between developers working on the client apps and backend also ensures that road blocks can be avoided and continual progress is achieved. The less time we spend guessing how the API is going to work in every situation, the more time we can spend building out the awesome features that make our apps unique.

Sign up to be notified of new blog posts and be the first to
vy Apps!

Email

SEND

104
Shares

BY: MATT TEA

Matthew Tea is a developer with a passion for qualit
with a strong desire to learn new and upcoming technologies.

RECOMMENDED ARTICLES

Can Core ML in iOS Really Do Hot Dog Detection Without Server-side Processing?

Machine learning has quickly become an important bedrock for a variety of applications.

Beyond Constraints: Crafting Advanced iOS Animations with Auto Layout

How we think about design at Savvy Apps encouraged us to develop a new animation

Keep Reading

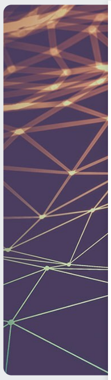
Keep Reading

104
Shares

to Start Android Development with an ackground

ou've ever done in the past is iOS opment, looking to build an app on id might make you feel like you're entering...

Keep Reading



How to App

What is creators to drive user engagement and simplify the app onboarding process. It's useful in tracking referrals and...

Keep Reading

Skip to a Section

RECENT ARTICLES

Building a Real-World Web App With Vue.js and Firebase

Jeremy Olson Talks About Success in the Evolving App Store

Using Advanced Auto Layout Techniques to Adapt Interfaces to Screen and Content

10 App Marketing Tips to Boost Your App's Launch Success

Choosing a Firebase Database For Your App: Realtime Database vs. Cloud Firestore

The Definitive Guide to Expanding Your Native A

The Imp

Skip to a Section

104
Shares

YOU MADE IT THIS FAR SO...

Want to work with us?

Savvy Apps is a Washington, D.C. mobile design and mobile development company serving global brands and cutting-edge startups. We're a product team for hire that's driven by making life better, one app at a time.

JUST SAY HI

OR REQUEST A QUOTE

LINKS

Careers

Support

Privacy Policy

Terms

CONTACT

1850 Centennial Park Drive
Suite 100
Reston, Virginia 20191

(703) 544-9191

See It On a Map

104
Shares

Skip to a Section