# RESTful API Design

- Our main focus in this course will be on building a ***RESTful API***, but before we talk about how to build a RESTful API, we need to understand what a RESTful API is.

- In the most technical sense, a RESTful API is an API that conforms to the ***REST*** (***REpresentational State Transfer***) architectural paradigm. Under the REST paradigm, clients use stateless, predefined operations to request specific resources from a server, and the server transfers some kind of representation of those resources back to the client.

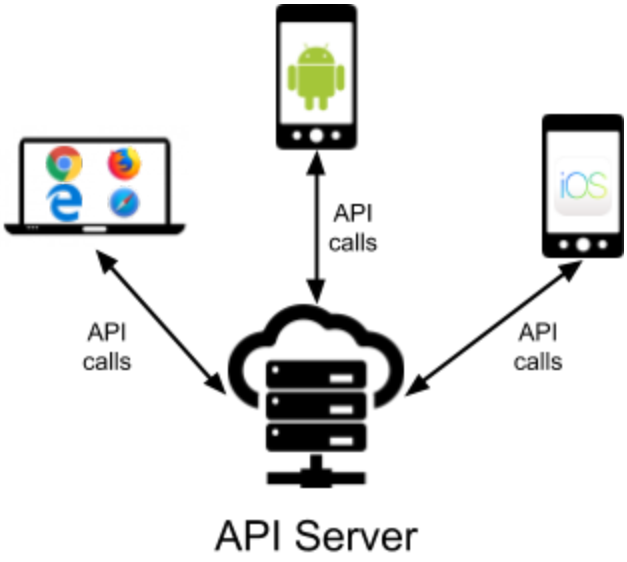
- There's a lot to unpack there. Let's begin to do so. First, let's focus on the "API" part of "RESTful API."

- An ***API*** (***application programming interface***) is simply a clearly-defined set of methods that allow one software component to communicate with another software component.

- In this course, we will specifically deal with web APIs that allow software on one machine (the client) to communicate with software on another machine (the server) via predefined methods.

- In particular, in our context, the server is our API server, which you can think of as the gateway to our web application's data, and the client is a program on another computer that wants to use that application data, e.g. a web browser on a laptop, an app on a mobile device, etc.

- This architecture looks something like this:




- We're specifically interested in RESTful APIs here. What does it mean for a web API to be RESTful?

- A RESTful API has three main elements:
    - **Resources** – These are the actual pieces of application data (i.e. the state; the "S" in "REST"). You can think of them as the "nouns" of the system. For example, if we were building an API to power an application like Airbnb in which guests could reserve lodgings for short-term rentals, the resources (i.e. the "nouns") would include guests, owners, lodgings, reservations, etc.
    - **Actions** – These are the way clients interact with resources (they transfer the state; the "T" in "REST"). You can think of them as the "verbs" of the system. For example, in our Airbnb-like API, there might be actions to list lodgings near a certain location and to book a lodging.
    - **Representations** – These are the way resources are presented to clients (and the "RE" in "REST"). Importantly, clients of a RESTful API are not provided with direct access to actual resources, only representations of them. For example, in our Airbnb-like API, we might store individual lodgings as rows in a database table, but when a client requests information about lodgings from our API, it might send a JSON-formatted list containing locations, prices, descriptions, etc. for the lodgings. This JSON list is the representation of the underlying resources.

- These three elements of a RESTful API are implemented using three different elements of the HTTP protocol:
    - Resources ↔ URLs
    - Actions ↔ HTTP methods (e.g. GET, PUT, POST, DELETE, etc.)
    - Representations ↔ Media types (e.g. `application/json`, `application/xml`, etc.)

- It is our job as API writers to define the resources, actions, and representations our API will use.

- Design is as important here as it is in any application with a user interface. For example, writing an API that simply exposes CRUD database operations (create, read, update, delete) is usually not a good design. Instead, our design should reflect the specific functionality needed by the client.

- For instance, in our Airbnb-like API, we likely want to include functionality to allow the client to book a lodging for a specific user. This functionality might involve modifications to many different places in the data (e.g. creating a new reservation, linking that reservation to the user who made it, updating the lodging to mark it as unavailable during the new reservation, etc.). It would be better for our API to have a single "reserve" action that does all of this work instead of individual actions that perform each of the individual data updates.

# Designing a very simple API

- Let's look at API design in more detail, starting with a simple example. Imagine we are writing a API that simply tells the client whether it is raining in any city in the US. How could we design this API?

- A good place to start would be to identify the nouns (resources) and verbs (actions) in the domain.

- What are the nouns in this case? Really, there is only one: "city".

- You might be tempted to say that something like "rain" is a noun in this situation, but really, whether or not it's raining is a piece of information that's associated with each specific city, so we'll treat it as such.

- What, then, are the verbs (i.e. the actions a client can take) in this case? Again, there really is only one: “check” (whether it’s raining in a specific city).

## Mapping resources to URLs

- So how do we take this information and turn it into an API specification? We can start by converting our nouns (the resources) to URLs. To do this, let’s briefly look at the components of a basic URL:

  ```
  http://www.example.com/some/specific/thing
  ```

- This URL can be broken down into the following three components:
  - **Scheme** – `http` – indicates the protocol used for the request being made.
  - **Host** – `www.example.com` – indicates the host to which the request is being made. You can think of this as indicating a specific server to which to make a request (although a single host name can actually correspond to many servers).
    - The host can also sometimes include a specific port number to which to make the request, e.g. `www.example.com:8080`.
  - **Path** – `/some/specific/thing` – identifies the specific resource on the host being requested.

- As you might suspect based on the breakdown above, we will use URL paths to represent the different resources (i.e. the nouns) in our API.

- In our case, we have one noun, city, and there are many resources of type “city”. In other words, we have a collection of cities.

- Under the REST paradigm, each individual item in a collection should be assigned a URL path based on some kind of identifier. In this case, there are many different kinds of identifier we could assign to US cities, but a natural one is ZIP code, so we’ll use this.

- The typical URL path format used to represent an individual resource within a collection looks like:

  ```
  /collection/itemID
  ```

- Thus, the URL path we'll use to represent each city resource in our API is:

  `/cities/{zip}`

  where `{zip}` is a placeholder for a specific city's ZIP code, e.g. `/cities/97330`.

- Note that we've used the plural "`cities`" here in our URL. It is common practice to use the plural version of a noun to indicate that it represents a collection.

## Mapping actions to HTTP methods

- Now that we've set up a URL scheme for our API's resources (i.e. its nouns), let's move on to set up actions to represent the verbs we identified.

- When specifying actions, we are restricted to using only documented HTTP methods, e.g. GET, POST, PUT, DELETE, etc. Each of these methods has specific semantics associated with it, e.g.:
  - **GET** – used to retrieve data from the server, i.e. to get the representation for a specific resource.
  - **POST** – used to provide data to the server. By convention, POST is used to create a resource whose representation is specified in the POST request's body.
  - **PUT** – used to create or replace a specific resource. The difference between PUT and POST is that PUT is ***idempotent***, i.e. calling it once or many times has the same effect. POST is not idempotent. For example, making the same POST request could submit an order multiple times.
    - A rule of thumb for differentiating between PUT and POST is that PUT should be used with a specific resource. For example, you would PUT to `/orders/12345`. POSTs, on the other hand, are usually made to a "factory" URL to create a new resource of that type. For example, you would POST to `/orders`.
  - **DELETE** – used to remove/delete a specific resource.

- Importantly, we can't define custom verbs/actions. Instead, we must define our nouns in such a way that we can use the predefined HTTP methods as verbs.
  - For example, in the context of the Airbnb-like API we discussed above, there is no verb "reserve", but using the noun "reservation", allows us to use HTTP POST to perform a "reserve" action by creating a reservation.

- Coming back to our rain API, we have only one verb: check (whether it's raining in a specific city). This verb maps nicely to the HTTP GET method. Specifically, checking whether it's raining in a specific city corresponds to retrieving the "is-raining" information about that city.

- In other words, our check action will correspond to making an HTTP request of the following form:

```
GET /cities/{zip}
```

## Representing resources with media types

- Finally, with our resource URLs and actions figured out, we need to determine how we will represent our resources.

- There are two places where a resource might need to be represented:
    - In the body of a response from our API.
    - In the body of a request to our API.

- By convention, GET requests should have empty bodies. Thus, the only place we need to represent resources in our rain API is in the bodies of responses to the GET request we formulated above.

- Our representation must use a specific ***media type***, which is simply a content format. Every media type has an associated two-part identifier. For example, JSON-formatted content has the media type identifier `application/json`.

- There are a few media types that are commonly used in conjunction with RESTful APIs, but, increasingly, JSON is the most popular, so we will stick with JSON here.

- The bodies of our API's responses to requests to the `/cities/{zip}` URL just need to indicate whether or not it is raining in the city corresponding to the specified ZIP code, so our representation can be simple. We'll use a JSON object with a single boolean value indicating whether or not it's raining, e.g.:

```
{ "raining": false }
```

- Note that this representation doesn't depend on whatever way we're storing data behind our API server. We could be storing individual city data as rows in a MySQL database table or as documents in a MongoDB database, or we might not be storing data at all and might instead simply be forwarding the query on to the Weather Underground API. The representation abstracts this all away to present the resource to the client in a way that is simple to understand and use.

- With our representation in place, our API is complete. Here is an example exchange with our API:

```
Request: GET /cities/97330
Response: { "raining": true }
```

# Designing an API for a more complex application

- The example above was a good way to get started thinking about how to design a RESTful API, but it was so simple that we didn't get to talk about some of the finer points of API design.

- Let's now look at a more complex example so we can dig a little more deeply. Specifically, let's work through designing an API for the Airbnb-like application we described above. Again, let's start by identifying the nouns/resources and verbs/actions associated with the application.

- We can identify two nouns/resources for this domain:
    - **Lodging** – any lodging (house, apartment, room, etc.) available to be reserved.
    - **User** – a user of the application. For simplicity, we combine the "owner" and "guest" roles under this single noun. Some users may own lodgings but make no reservations, some users might own no lodgings and only make reservations, and some users might both own lodgings and make reservations.

- And, we can identify the following verbs/actions:
    - **List (lodgings)** – a user views a list of lodgings
    - **View (single lodging)** – a user views information about a specific lodging
    - **Add (lodging)** – a user adds a new lodging they own
    - **Modify (lodging)** – a user modifies a lodging they own
    - **Remove (lodging)** – a user removes a lodging they own

    - **Reserve** – a user reserves a specific lodging
    - **View (single reservation)** – a user views information about a single existing reservation
    - **Modify (reservation)** – a user modifies an existing reservation
    - **Cancel (reservation)** – a user cancels an existing reservation
    - **List (a user's reservations)** – a user lists their entire history of reservations
    - **List (a user's lodgings)** – a user lists all of the lodgings they own
    - **List (a lodging's reservations)** – a user lists all of the reservations for a particular lodging

- There are probably more nouns and verbs we might want to use to create a full-featured application, but these will get us started.

- Our approach to designing this more complex API will be to step through each of the verbs we listed above and answer the following questions:
    - What is the associated noun/resource, and what URL should we use to represent this resource?
    - What HTTP method should we use to represent this verb/action?
    - What resource data should do we need to have included in the body of the request associated with the URL and HTTP method identified above?
    - What resource data should we return in the body of the response associated with the URL and HTTP method identified above?

- After answering these questions for each of the actions listed above, we should have a complete API design.  Let's get started!

## List (lodgings)

- **What is the associated noun/resource, and what URL should we use to represent this resource?**
    - The resource here is the entire collection of lodgings.  For dealing with the entire collection, we'll simply use this URL:

      ```
      /lodgings
      ```

- **What HTTP method should we use to represent this verb/action?**
    - In this case, we want to retrieve information from the API, specifically a list of lodgings. The HTTP GET method is the right one to use here.

- **What resource data should do we need to have included in the body of the request associated with the URL and HTTP method identified above?**
    - In this case, we don't need any data in the request body. We are simply using the GET method to fetch a list of lodgings.

- **What resource data should we return in the body of the response associated with the URL and HTTP method identified above?**
    - We'll use JSON to represent the content of all requests and responses in our API. We want to return a list of lodgings in the response to this API call. It's common practice to wrap all response data within a containing JSON object, to which we'll later be able to add various metadata about the response, so we'll do that here. We won't worry for now about the specific data we'll return about each individual lodging in the response, but we'll represent them as JSON objects. Our entire API call, then, will look like this:

```
Request: GET /lodgings
Request body: -
Response body:
  {
    "lodgings": [
      {
        "id": 12345,
        "description": "A nice place to stay",
        "address": "...",
        ...
      },
      {
        "id": 23456,
        ...
      }
    ]
  }
```

## View (single lodging)

- **What is the associated noun/resource, and what URL should we use to represent this resource?**
    - The resource here is a single lodging. As above in our rain API, we can use a URL here of the form `/collection/itemID`:

      `/lodgings/{lodgingID}`

- **What HTTP method should we use to represent this verb/action?**
    - Here again, we want to retrieve information from the API, specifically info about a single lodging. HTTP GET is again the appropriate method here.

- **What resource data should do we need to have included in the body of the request associated with the URL and HTTP method identified above?**
    - Again, we don't need any data in the request body. We are simply using the GET method to fetch info about a single lodging.

- **What resource data should we return in the body of the response associated with the URL and HTTP method identified above?**
    - Here, we'll return a JSON object containing info about the specified lodging. If needed, this object can contain response metadata. Our entire API call will look like this:

```
Request: GET /lodgings/{lodgingID}
Request body: -
Response body:
  {
    "id": 12345,
    "description": "A nice place to stay",
    "address": "...",
    ...
  }
```

## Add (lodging)

- **What is the associated noun/resource, and what URL should we use to represent this resource? AND, what HTTP method should we use to represent this verb/action?**
    - The resource here is again a single lodging, but in this case, we don't yet have an ID for the lodging. Instead, we want to create a new lodging (with a new ID) and add it to the collection of lodgings. In a situation like this, it is conventional to make an HTTP POST request to the URL representing the *collection* within which we want to create a new item:

      ```
      POST /lodgings
      ```

- **What resource data should do we need to have included in the body of the request associated with the URL and HTTP method identified above?**
    - In this case, we do need a request body that specifies all of the information needed to create a new lodging. We'll use a JSON object.

- **What resource data should we return in the body of the response associated with the URL and HTTP method identified above?**
    - We don't really need a response body here. If the POST request succeeds, the client will know the lodging was created. (We'll see later how to use HTTP status codes to indicate the success/failure of a particular request). Our entire API call will look like this:

      ```
      Request: POST /lodgings
      Request body:
        {
          "description": "A nice place to stay",
          "address": "...",
          ...
        }
      Response body: -
      ```

## Modify (lodging)

- **What is the associated noun/resource, and what URL should we use to represent this resource?**

    - The resource here is a single *existing* lodging, so we'll use this URL:

      ```
      /lodgings/{lodgingID}
      ```

- **What HTTP method should we use to represent this verb/action?**
    - When modifying a resource, it is conventional to use HTTP PUT.

- **What resource data should do we need to have included in the body of the request associated with the URL and HTTP method identified above?**
    - When using the HTTP PUT method, the request body will be used to *replace* the existing resource. In other words, the request body for a PUT request needs to specify the *entire* resource. Thus, here our request body will need to contain the same information as our POST body above.
        - If you want to allow an API call to make a *partial* update, you should use the HTTP PATCH method. In this case, the request body can contain only a set of *changes* to be made to the existing resource.

- **What resource data should we return in the body of the response associated with the URL and HTTP method identified above?**
    - Again, we don't really need a response body here. If the PUT request succeeds, the client will know the lodging was updated. Our entire API call will look like this:

      ```
      Request: PUT /lodgings
      Request body:
        {
          "description": "A very nice place to stay",
          "address": "...",
          ...
        }
      Response body: -
      ```

## Remove (lodging)

- **What is the associated noun/resource, and what URL should we use to represent this resource?**
    - The resource here is again a single *existing* lodging, so we'll use this URL:

      ```
      /lodgings/{lodgingID}
      ```

- **What HTTP method should we use to represent this verb/action?**
    - Here, we want to *remove* a resource. HTTP DELETE is the appropriate method to use to do this.

- **What resource data should do we need to have included in the body of the request associated with the URL and HTTP method identified above?**
    - We don't need any data in the request body for this DELETE call. The lodging ID specified in the URL provides all the information we need to remove the lodging.

- **What resource data should we return in the body of the response associated with the URL and HTTP method identified above?**
    - Here, too, we don't need a response body. If the DELETE request succeeds, the client will know the lodging was updated. Our entire API call will look like this:

      ```
      Request: DELETE /lodgings/{lodgingID}
      Request body: -
      Response body: -
      ```

## Reserve

- **What is the associated noun/resource, and what URL should we use to represent this resource? AND, what HTTP method should we use to represent this verb/action?**
    - The questions about what the resource and method are here are interesting. In particular, there is no HTTP method to "reserve". As we described above, in a situation like this, we can actually create a noun/resource that allows us to use existing HTTP methods. In this case, instead of performing a "reserve" action on a lodging resource, we'll use a new resource "reservation". The "reserve" action, then, will create a reservation.

      Now, we have a situation similar to above when we added a new lodging. In particular, to create a reservation, we can make an HTTP POST request to the URL representing the collection of reservations:

      ```
      POST /reservations
      ```

- **What resource data should do we need to have included in the body of the request associated with the URL and HTTP method identified above?**
    - In this case, we do need a request body that specifies all of the information needed to create a new reservation. We'll use a JSON object.

- **What resource data should we return in the body of the response associated with the URL and HTTP method identified above?**
    - Again, we don't really need a response body here. If the POST request succeeds, the client will know the lodging was created. Our entire API call will look like this:

```
Request: POST /reservations
Request body:
  {
    "lodgingID": 12345,
    "startDate": "...",
    "endDate": "...",
    ...
  }
Response body: -
```

## View (single reservation)

- This API call will look very similar to the one we designed for viewing a single lodging. Here's what the whole API call will look like:

```
Request: GET /reservations/{reservationID}
Request body: -
Response body:
  {
    "id": 12345,
    "lodgingID": 23456,
    "startDate": "...",
    "endDate": "...",
    ...
  }
```

## Modify (reservation)

- Again, this API call will look very similar to the one we designed for modifying an existing lodging. Here's what the whole API call will look like:

```
Request: PUT /reservations/{reservationID}
Request body:
  {
    "lodgingID": 23456,
    "startDate": "...",
    "endDate": "...",
    ...
  }
Response body: -
```

## Cancel (reservation)

- Canceling an existing reservation is the same as deleting it, so we can use an HTTP DELETE here. Here's what the whole API call will look like:

```
Request: DELETE /reservations/{reservationID}
Request body: -
Response body: -
```

## List (a user's reservations)

- **What is the associated noun/resource, and what URL should we use to represent this resource?**
    - In this case, we're actually dealing with two different types of resource: a user and that user's reservations. There are various ways we could formulate a URL scheme to facilitate accessing resources that are related in this way. A common one is is to use a URL of the form `/collection/itemID/collection`. We'll do that here, assuming we have a `users` collection:

      `/users/{userID}/reservations`

    - Importantly, it is usually best to avoid using URLs more complex than `/collection/itemID/collection`. For example, using a URL like `/users/{userID}/reservations/{reservationID}` would be considered bad practice. We'll see later how we can use a concept called HATEOAS to eliminate the need for complex URLs.

- **What HTTP method should we use to represent this verb/action?**
    - In this case, we're just retrieving information from the API. HTTP GET is the right method to use here.

- **What resource data should do we need to have included in the body of the request associated with the URL and HTTP method identified above?**
    - As usual, we don't need a request body when making a GET request.

- **What resource data should we return in the body of the response associated with the URL and HTTP method identified above?**
    - Here, we'll simply return a list of the user's reservations. Our entire API call, then, will look like this:

```
Request: GET /users/{userID}/reservations
Request body: -
Response body:
  {
    "reservations": [
      {
        "id": 12345,
        "lodgingID": 23456,
        "startDate": "...",
        "endDate": "...",
        ...
      },
      {
        "id": 34567,
        ...
      }
    ]
  }
```

## List (a user's lodgings)

- This API call will mirror the one we just created for listing a user's reservations:

```
Request: GET /users/{userID}/lodgings
Request body: -
Response body:
  {
    "lodgings": [
      {
        "id": 12345,
        "description": "A very nice place to stay",
        "address": "...",
        ...
      },
      {
        "id": 23456,
        ...
      }
    ]
  }
```

## List (a lodging's reservations)

- Now that we've seen how to use a URL of the form `/collection/itemID/collection`, designing this API call is straightforward. It will look like this:

```
Request: GET /lodgings/{lodgingID}/reservations
Request body: -
Response body:
  {
    "reservations": [
      {
        "id": 12345,
        "lodgingID": 23456,
        "startDate": "...",
        "endDate": "...",
```

```
      ...
    },
    {
      "id": 34567,
      ...
    }
  ]
}
```

- With this API call in place, we have our API completely designed!

# HTTP status codes

- In addition to the response bodies described above, ***HTTP status codes*** are an important way the server can indicate to the client what happened with a request.

- An HTTP status code is simply a numerical value that is attached to an HTTP response to indicate the status of a request. In fact, *every* HTTP response *must* have an HTTP status code.

- Choosing the correct status code for a response is important, so let's spend some time looking at how to do this. Once we do, we should be able to return the correct status code with all of our responses (including errors).
    - Often, responses with error status codes will also include a response body containing more information about the error.

- Importantly, every status code is a 3-digit number whose first digit indicates a broad category within which the status code falls. We'll examine each of these categories in turn.

## 2xx – Success

- Status codes that begin with a 2 indicate that the request succeeded in some way. The most important 2xx status codes are:
    - **200** – **OK** – used to indicate a successful request, most often when the response also includes a body
    - **201** – **Created** – used to indicate that a request was successfully created, e.g. as a result of a POST request
    - **204** – **No Content** – used for successful requests with no response body

## 3xx – Redirection

- Status codes that begin with a 3 are used to let the client know that the requested resource is located somewhere different than the requested URL. The most common 3xx status codes are:
    - **301** – **Moved Permanently** – used to let the client know the requested resource has been permanently moved to another URL
    - **302** – **Found** – used to let the client know the requested resource has been *temporarily* moved to another URL

## 4xx – Client error

- Status codes that begin with a 4 are used to let the client know that their request failed because there was something wrong with it. In other words, 4xx status codes indicate a client error. The most common ones are:
    - **400** – **Bad Request** – used to flag requests with invalid or malformed data
    - **401** – **Unauthorized** – used to flag requests with the wrong authentication credentials for the requested resource
    - **403** – **Forbidden** – used to flag requests where the (possibly-authenticated) user does not have permission to access the requested resource
    - **404** – **Not Found** – used to flag requests for resources that do not exist (or that are hidden from the requesting user)
    - **405** – **Method Not Allowed** – used to indicate that a particular HTTP method can't be used in conjunction with the requested resource

## 5xx – Server error

- Finally, status codes that begin with a 5 are used to indicate that something went wrong on the server *not* because of client request data. 5xx errors should only be used for exceptional and unexpected cases (e.g. a broken database connection). The 5xx status code most commonly returned by an API is:
    - **500** – **Internal Server Error** – used to indicate that an unexpected condition prevented the server from fulfilling the request

# Hypermedia as the engine of application state (HATEOAS)

- When a client makes an API call, it will often want to follow up with a related API call. For example, after making a call to view information about a specific lodging, the client might want to make a follow-up call to get information about that lodging's owner.

- To do this, the client could pull the lodging's owner's user ID out of the first API response (if it's present there), use that user ID construct a URL for an API call to retrieve information about the user, and then execute that API call.

- This approach is somewhat brittle. First, it relies on the lodging's owner's user ID being present in the response to the API call that fetches information about a single lodging.

- Second, it relies on the programmer of the client knowing the URL to use to get information about a user given the user's ID. Even if they know this URL, it assumes the API will not change.

- A better approach here is to employ the principle of ***hypermedia as the engine of application state*** (or just ***HATEOAS*** for short). Under this principle, each API request should return links that allow the client to directly find related resources.

- This principle is similar to the principle that makes the web useful. When you visit a web page in a browser, that web page will usually contain links to related web pages. You can simply follow these links to get to related information. Similarly, when using an API that follows the HATEOAS principle, an API client can simply follow links in the response to reach other related resources.

- Following the HATEOAS principle, the first API call in our example above (to fetch information about a specific lodging) might return a response like this:

```
{
  "id": 12345,
  "description": "A very nice place to stay",
  "address": "...",
```

```
  ...,
  "links": [
    {
      "rel": "owner",
      "href": "/users/98765",
    },
    {
      "rel": "reservations",
      "href": "/lodgings/12345/reservations"
    },
    ...
  ]
}
```

    - Note here that the "`rel`" field indicates the "relationship" the link corresponds to. This is conventional terminology.

- In this way, if the client wants to follow up by requesting information about the owner of the lodging, it can simply extract and use the relevant URL.

- Importantly, HATEOAS links can occur at any level of a hierarchical response body. For example, in the response to a GET request to our `/lodgings` URL, each individual lodging object might contain its own set of related links.

# Pagination

- Some kinds of requests to an API might generate many results. For example, a GET request to our API's `/lodgings` URL might generate a list containing of hundreds or thousands of individual lodgings.

- Sending such a large response back to the client could have a negative impact on the performance of both our server and the client, so we usually want to avoid doing this.

- To avoid this problem, it is good practice for an API to ***paginate*** large responses, that is, to break them down into many smaller "pages".

- For example, it would be a good idea for our API to paginate responses to GET requests to `/lodgings`.

- The first response to such a request might look like this:

```
{
  "page_number": 1,
  "total_pages": 127,
  "page_size": 10,
  "total_count": 1264,
  "lodgings": [ ... ]
}
```

- Then, to get each successive page, the client could append a ***query parameter*** to its request URL indicating the specific page of lodgings it is requesting, e.g.:

```
GET /lodgings?page=2
```

- In fact, when performing pagination at the API level, it is a good idea to use HATEOAS to include links to related pages, e.g.:

```
{
  "pageNumber": 1,
  "totalPages": 127,
  "pageSize": 10,
  "totalCount": 1264,
  "lodgings": [ ... ],
  "links": {
    "nextPage": "/lodgings?page=2",
    "lastPage": "/lodgings?page=127"
  }
}
```

- The databases we will look at in this course make it easy to implement pagination.

# Authentication

- Finally, it is worth mentioning the issue of authentication. Many of the API calls we designed above will need to be authenticated. For example, we may only want to allow a user to look at their own reservations and not the reservations of other users.

- Thus, our `/users/{userID}/reservations` API endpoint will need to use some form of authentication to verify that the user who initiated the request is allowed to see the requested reservations.

- It is also possible that our API might return *different* information for a particular API call depending on the user's authentication status. For example, our API might return different information in response to a GET request to the `/lodgings/{lodgingID}` endpoint if the requesting user is the *owner* of the specified lodging.

- Several other of our API endpoints will likely also require authentication of some form. Spend a minute to identify which ones might fall into this category.

- We will discuss how to implement authentication later in the course.