Topic: Hbnb documentation

Table of Content:

1) Introduction

2) High-Level Architecture

3) Package Diagram

4) Business Layer Logic

5) Class Diagram

6) API interaction & Sequence Diagram

Introduction

Welcome to the very brief and concise documentation for the HBnb project of this trimester. As part of our learning experience, our desire is to provide you with a seamless experience towards booking your very own destination for your next comfortable and adventurous stay at a Bed and Breakfast, brought to you by Cohort 26. When creating this blueprint, our desire was to take away all of the abstraction that comes with programming the application into functioning as desired, and simply make a roadmap that explains every concept of this application in proper detail, from the construction of the platform to the user interaction and relation to every component. In theory, this part of the project would be something that should contain much less, since it serves as a foundational document. I want to impress, however, that the foundation of this website can easily crumble if at any point this is made too brief. The construction of any house(or app) relies upon the ground being solid and firm. And thus, this document was born.

As mentioned, our mission is to provide a seamless experience for the user to interact with the application and its various components. But, what is it composed of? This document contains a couple of diagrams to better visualize everything that is happening as the user interacts and as that user input is sent through various entities that make up the functionality. More on them later. To better dive into how it works, we should work in phases and hop onto the first one:

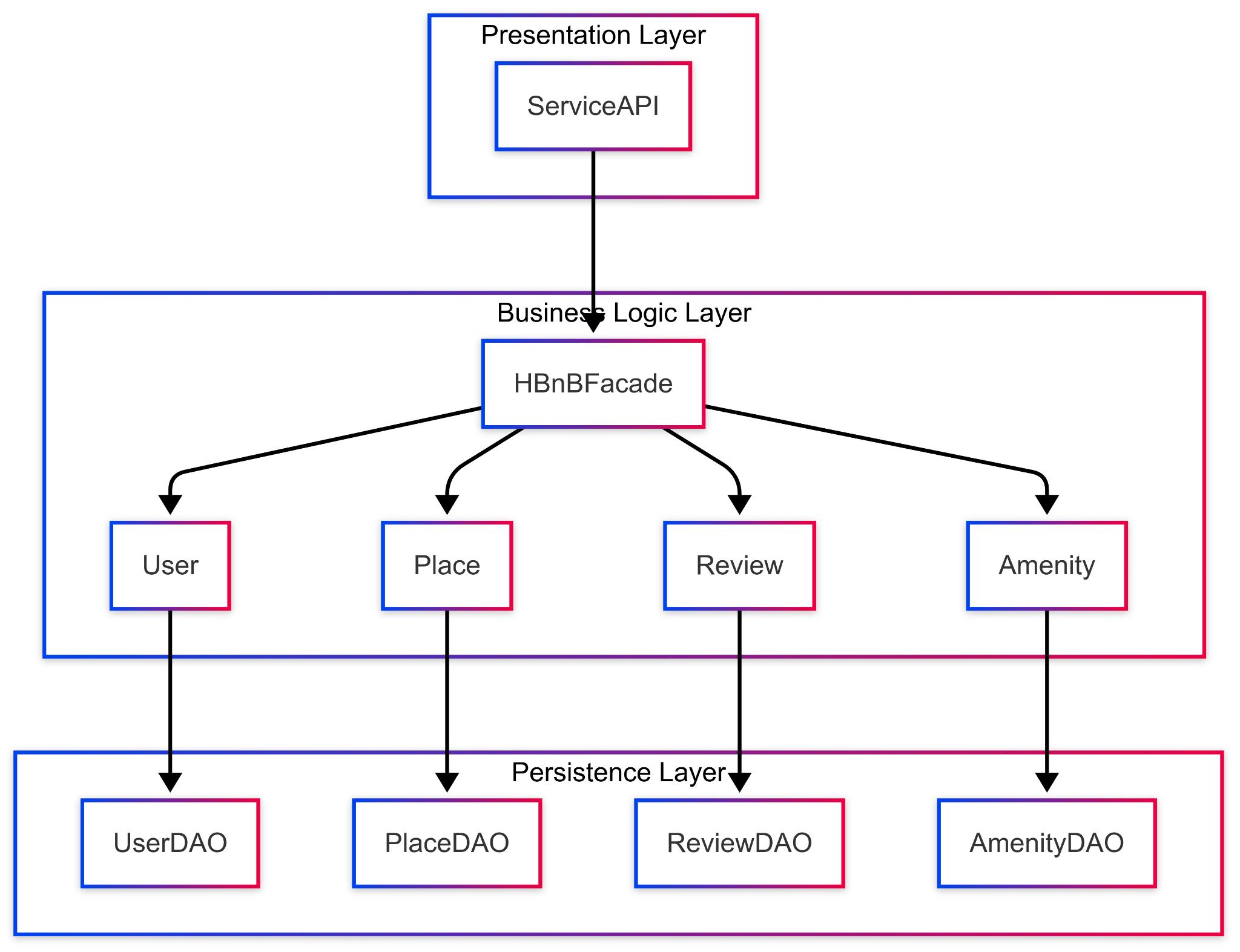
High-Level Architecture

First user interactions:

As with any application, that begins operation as soon as it is open, here we hop onto the very first concept. This one proves to be the most important: partly because without having a user, the application would serve little to no purpose, and mostly because this very segment serves as the most important part of the app's building ground: the service API. This server API is the first checkpoint of the application that serves the following purposes: it catches the eye of the user, it properly segments everything into a bundle of multiple functioning parts and exists as the agent by which information is received and delivered correctly. I find this last part to be the most important due to the fact that this feat of the server API is much less of a benefit and more so as the protocol that visually cues the customer of the Hbnb to interact with the app and presents the correct segmentation and modulation that happens in the back end area. Think of this section as the very first part of picking up the phone and being met with an enthusiastic voice prompt that invites you into the business atmosphere and then bombards you with a series of options that you can choose in order to receive the adequate service or to speak to the correct agent to provide you with such service. This frontline entity is tasked with engaging the customer in a very direct way; it receives orders and information from the person requesting the services and directs our customer to the correct area for the specific service or task requested. Despite appearances, this portion of our app does some significant things when set in motion: it prompts our customer to shift its attention individually to every segment that he or she wishes to explore in the application, and more importantly, it brings forth the methodology by which everything is individualized. "What could it be?", you might ask. Well, let’s be a little more particular from this moment on. For you see, now its time we discuss a little more in detail what these "interactions" are.

As a BnB, we want a customer to search for a place and stay in it. Now, that is as simple as it can get. What is suddenly not so simple is how everything must comply with a set of standards that make this a more complete experience other than just making a simple transaction. And that is where the particular lies, that everything must be particularly made for a task to perform. We set our sights for making the first concepts of the application with this idea that everything needed to exist as a particular and individual thing. Well, that is where our second agent comes into play: the business Facade. We talked about how our frontline agent, the Service API, visually set everything apart for the customer to interact with it more directly while also still putting up the complete piece of the puzzle. Well, this facade is where our agent begins to coalesce into more unity while still adhering to this aforementioned unity. Where the server was the teleprompter driving your attention to the required area you wanted to start service, the facade is actually the part that puts you in communication with the entity you wish to interact with. And it is true to say now that everything begins to set itself more apart, for you see: as the customer is now redirected to the particular entity with which it wants to, or needs to, be in touch with, the experience truly begins. But before we begin to unravel more of these intricacies about this app, it would do great service to better explain the purpose of the facade from the server's standpoint, as the facade exists to aid the server in an orderly fashion and to better service our customer's need. How, you say? Well, having that subtle sense of separation better glues the operation, and that is exactly what the facade does. As our customer interacts more directly with the smaller subcomponents of the big application, having more independent moving parts under separate supervision better ties the function of the app. As another analogy, think of a clock with various cogs, corks and screws, all independent but still moving in unison to do one thing: keep a clock ticking. The facade has multiple operations that allow our smaller cogs to better aid the customer to perform that one thing he wants but through various different things, and also allows the system to keep better track of the function of the app individually, but arranged as a whole. It works well for the app to perform as various moving pieces, so that one part does not rely upon a part for it to begin functioning and then performing its correct task. Instead, we should allow these cogs to move to its assigned task to accomplish a desire; and as such, if these cogs would require removing and reworking, it does not alter the functionality of the rest of the app.

Package Diagram:



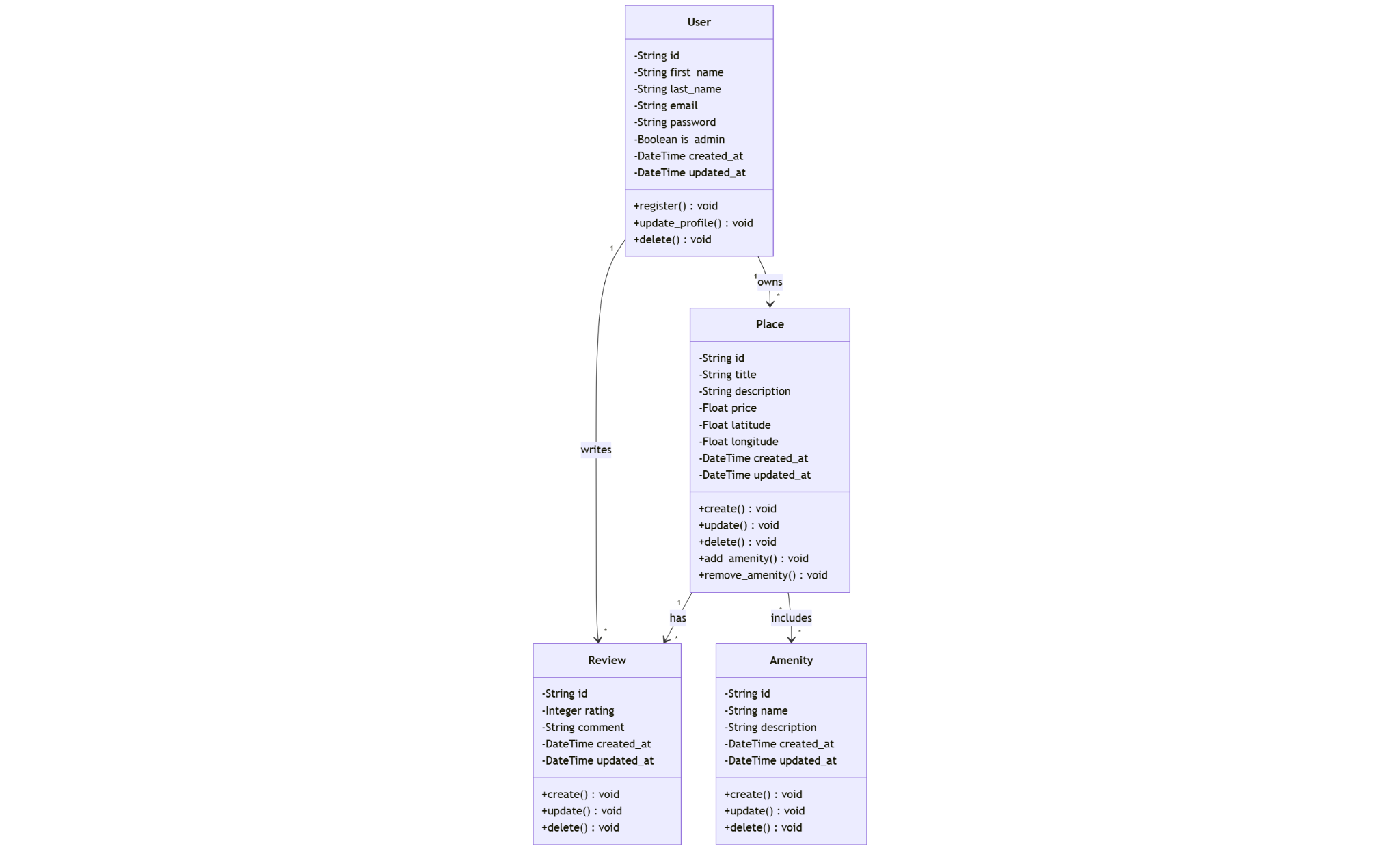
All of this better explains the need for the creation of the following diagram, where we designed it for the purpose of overviewing our bigger picture, by showing in theory how the app is expected to perform. The first part of the diagram represents the very first thing that the customer sees when they enter the application, and from there, everything begins to flow down as smoothly as it possibly can. As we flow down from the service API into the HBnb facade, it's important to note that the interface is assumed to work correctly and show different options to perform: be it searching for a place and/or booking it, creating or updating an account, leaving a review, etc. Going back to the teleprompter analogy, as the customer is redirected to its destination, we see the facade being to work. Take this as the moment in which the prompter says "One moment, please, as we transfer you". This is where the crux of this diagram starts to make sense, for you see, this redirection is the process of the server's communication with the according entity assigned with providing the app's function per the customer's request. These entities, as shown above, are the four basic pillars that make up the application: User, Place, Amenity and Review; all of which perform their own specific task and handle their own unique data and instruction, which is linked customer to server and server to database. So when a customer chooses to interact with his account, what the server has received as an instruction is to work with the entity known as user, and sends an order for the facade to keep contact between the customer and this entity, it does so by way of handling that interaction. If for example, a customer wants to create a user or update it, the facade does so by using the according redirect, which in the diagram is shown through the 'Handle\_user\_creation' redirect function that acts as the bridge between the server and customer. As the link is established, the customer gains access to said User entity and can now create, update or delete an account. That request is then parsed to the required functions that are in charge of the according operation and is stored in the database through DAO functions that are stored within each entity. It is all then fed from database unto server end and then to customer in a safe manner, by encrypting and decrypting data as the information travels from point a to point b.

Business Layer Logic:

Understanding up to which point everything must be separate, we now must explain why and where this part of the design requires unity. So far, we mentioned the existence of these entities and how they are meant to exist as individually as they possibly can. Well, there should also come a point when all of them come together to glue this, so as to not leave the app in a state of disarray, with each entity being its own thing, incapable of being cohesive. We have understood thus far that entities need to fulfill their own purpose to interact correctly with the user and the data being fed and received to the user of the application. We have done the first important task of this project, and that is to show how connecting between the user and the primitive parts of the application is possible. This diagram, however, is only the beginning of the operation, since there is yet a user to book a place and separate it as an actual order to be received by the host. If data was left to work separately, a common goal would not be achieved; instead, the lingering "what now?" interrogative would flow out and about, with no seeming finality being set in motion. And so, there must now exist a driven purpose behind all of these entities, a cue to make them perform as prescribed. In fact, it is now at this moment that the entity's purpose is worthy of explanation. And what better way to do it than to, once again, focus on the application’s goal.

This is what the entities exist for, to fulfill our customer needs. And it does this by adhering to its purpose, which is clearly defined by the data that it stores. User is meant to handle all matters and affairs regarding the management of users, their profile, credentials, their individual roles in the app, and so on. ‘Place’ exists so that we manage the storage of information regarding the places that are the focal point of our BnB; that we may allow places to be booked for getaway destinations by other users sitewide. As part of this role, the data stored in this entity concerns the data of this place's location, photographs, description, owner(s), amenities and reviews. Despite it being a primary component contained within the place entity, the amenity exists as an entity of its own. Despite its existence being dependent on the place, the manner in which we store the data regarding amenities shows the requirements for these entities to be created. See, in contrast to a location or a photo, which exists as a single type of data to be stored(a float data type and a jpg file, respectively) the amenity is not one single component that can be summarized as we would the aforementioned components of the place entity. Furthermore, entities would also require for there to be methods to be created, updated or deleted, and methods by which they are directly related to other entities. Case in point, the amenity is an entity whose existence must be structured with multiple data such as: a description, photograph, quantity, and methods that link it to its particular place. In fact, it is under this same guideline that we must now explore the final main entity, the review. Same as with the amenity, the entity is one that exists as a by-product of relation and dependance with the place. And while seemingly straightforward, this entity requires data and methods by which they are created, updated, linked to the places and the amenities of the places, etc.

Class Diagram:



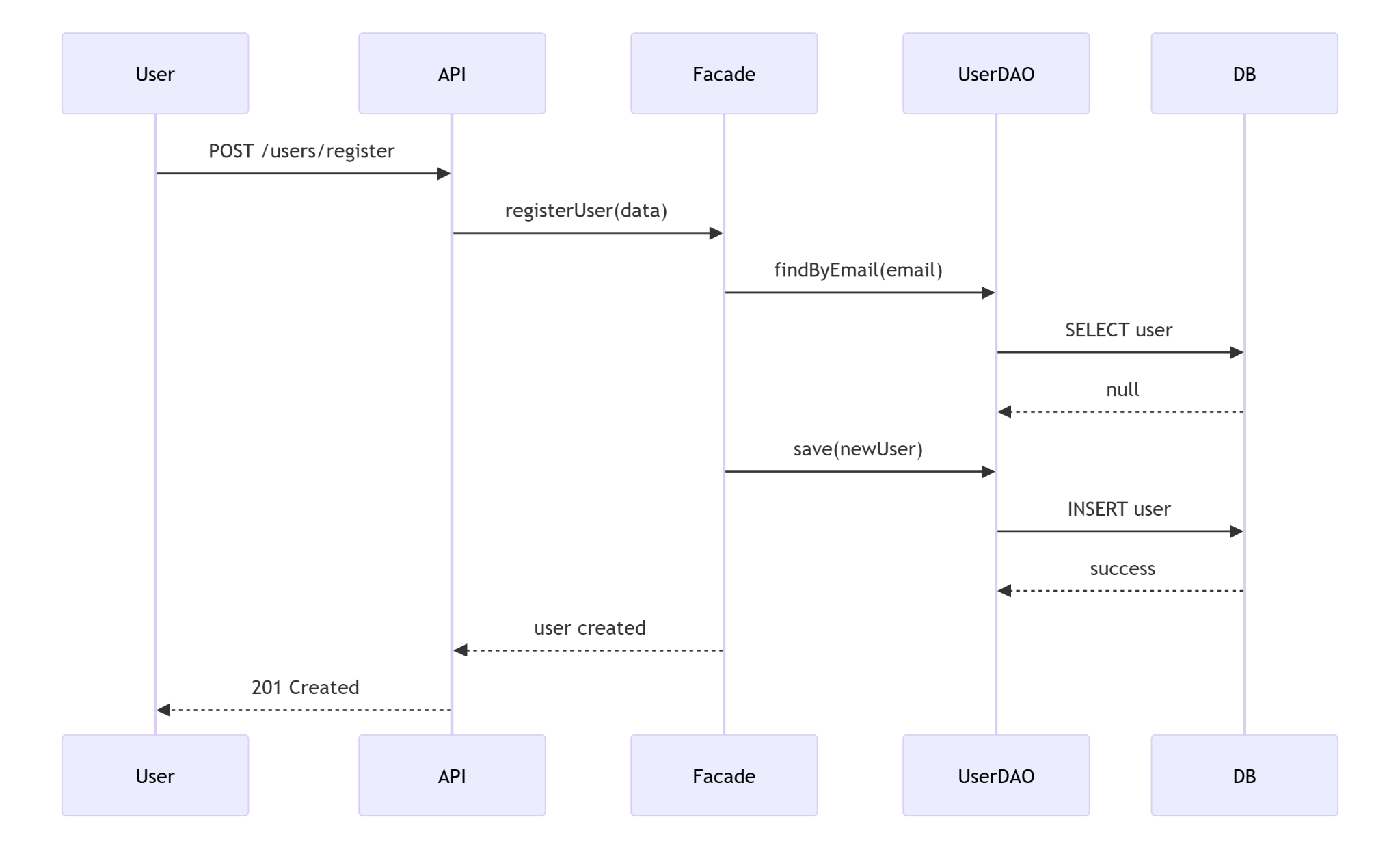
Now, so far in this discussion about entities and their components, one key aspect seems to reappear: interrelation. Between what? Well, the entities themselves. The class diagram visually explains the relation between classes, why they must relate, why the application cannot go on with this interrelation, and finally, more importantly, how they are related to each other in terms of data. The diagram links everything together based on theoretical functionality and design purpose. What does it mean by this? We can get into context now and provide a clear picture of how we expect this interaction to flow: when we enter the app, we have established that the user is meant to be created so that it may search for a destination of their choosing. Right away, we have the very first relation established: a ‘User’ of the app who looks for a place. However, before this interaction takes place, a ‘Place’ must be established in a database for others to find. And this itself, prompts another previous user to upload this destination to the database. After all, we must think in terms of the application being a purveyor of service. Which leads, of course, for the second relation to manifest: the relation between Place to a User. This kind of relation lets us know that a place is tied directly to a specific user of the site, who also manages it and reviews all the things concerning the intricacies of the place, such as the description, location, amenities, reviews, dates for booking, etc. We can also see how users can perform two additional actions from separate standpoints: a guest user can leave a review, which is linked to a place and its services and amenities, for others to review. Likewise, an administrative user can also modify the place's amenities, add them, remove them, etc.

One key thing to note is that, for the purpose of the diagram, what has been included were the concrete relation between entities. While the abstract connections between entities are assumed, they are overlooked since it is inferred to happen. The type of relation all of these entities have are distinct, but very much a by product of design purpose. For example, an amenity can independently exist, but it would probably serve no purpose if it was not linked to a place in particular. A review exists almost primarily to enhance experience in the app, but any purpose other than to complement a place cannot be imagined. It's not to mean that their worth is inexistent, however. From the customer's standpoint, a place would become unappealing to a customer if it were not complimented by the amenities and by reviews that accompany them. It makes sense that there is an association from the amenity to the place and from the review to a place, and also how a user is linked to an amenity and to a review, since these would only be created and set up for evaluation by users.

And like this, more structure has been given to this grand object that was created. By structure, it should be noted that the correct term implied here is contextualization. It looks much less like a brainstorm; instead, correct terminology and unity has been added, as far as this part of the structure can be explained, that is. What our previous diagrams fail to make is the one thing that warrants the creation of another set of diagrams, which should standardize and structure the sequence of events by which operations are performed and how correct dialogue between user and server/data is established. Hence, the sequence diagrams. Our clear goal is to better show this unity and interaction as a chain of events, highlighting how these interactions unfold and how data is manipulated by the server and the user. This section unifies what was explained in both diagrams, and our goal is to highlight exactly that. It must not be understated that user experience is the goal established, and these diagrams are designed to go into more detail about how interclass relations unfold.

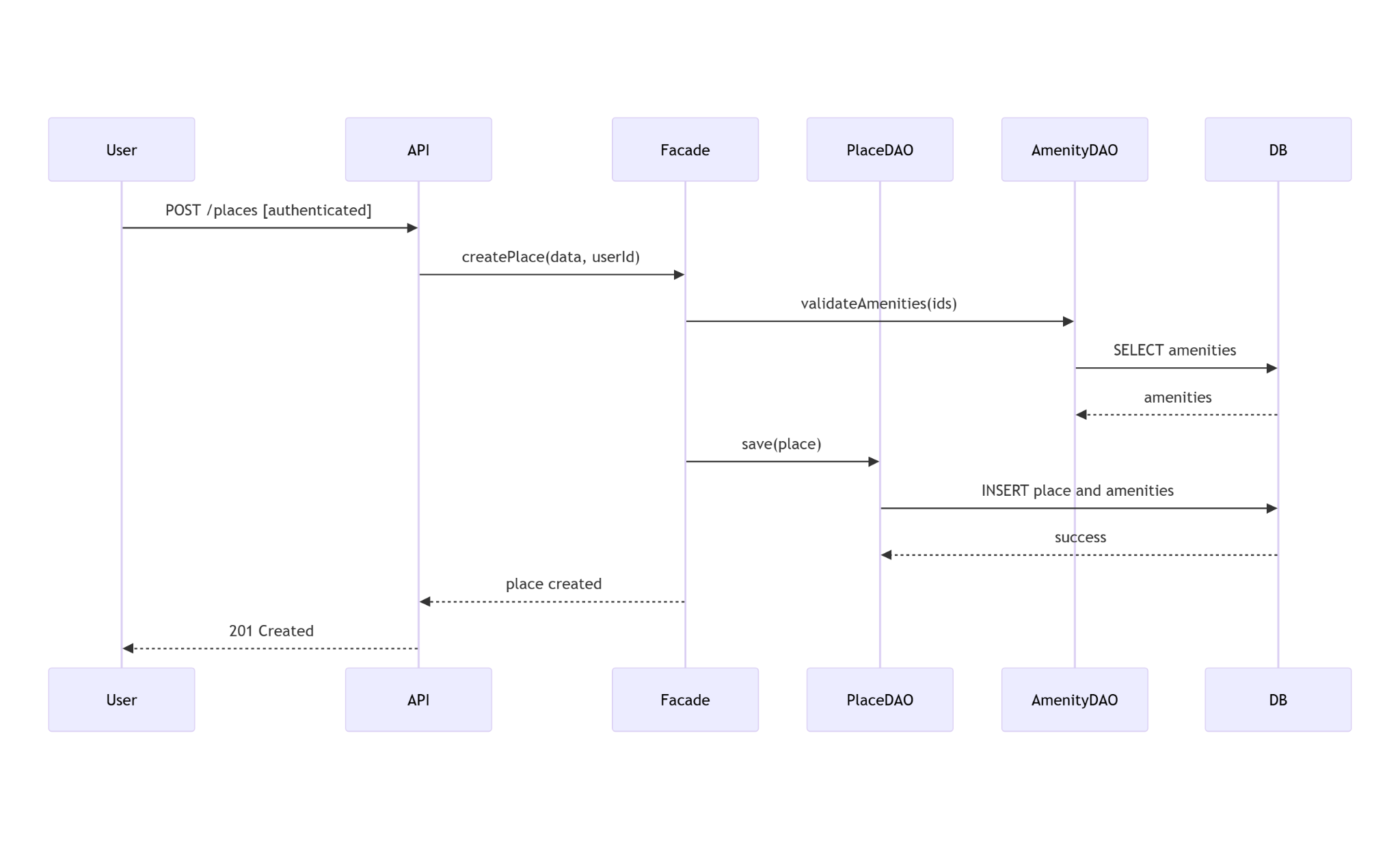
API Interaction & Sequence Diagrams:

Let’s begin with the first step, which is the creation and the manipulation of the user data.

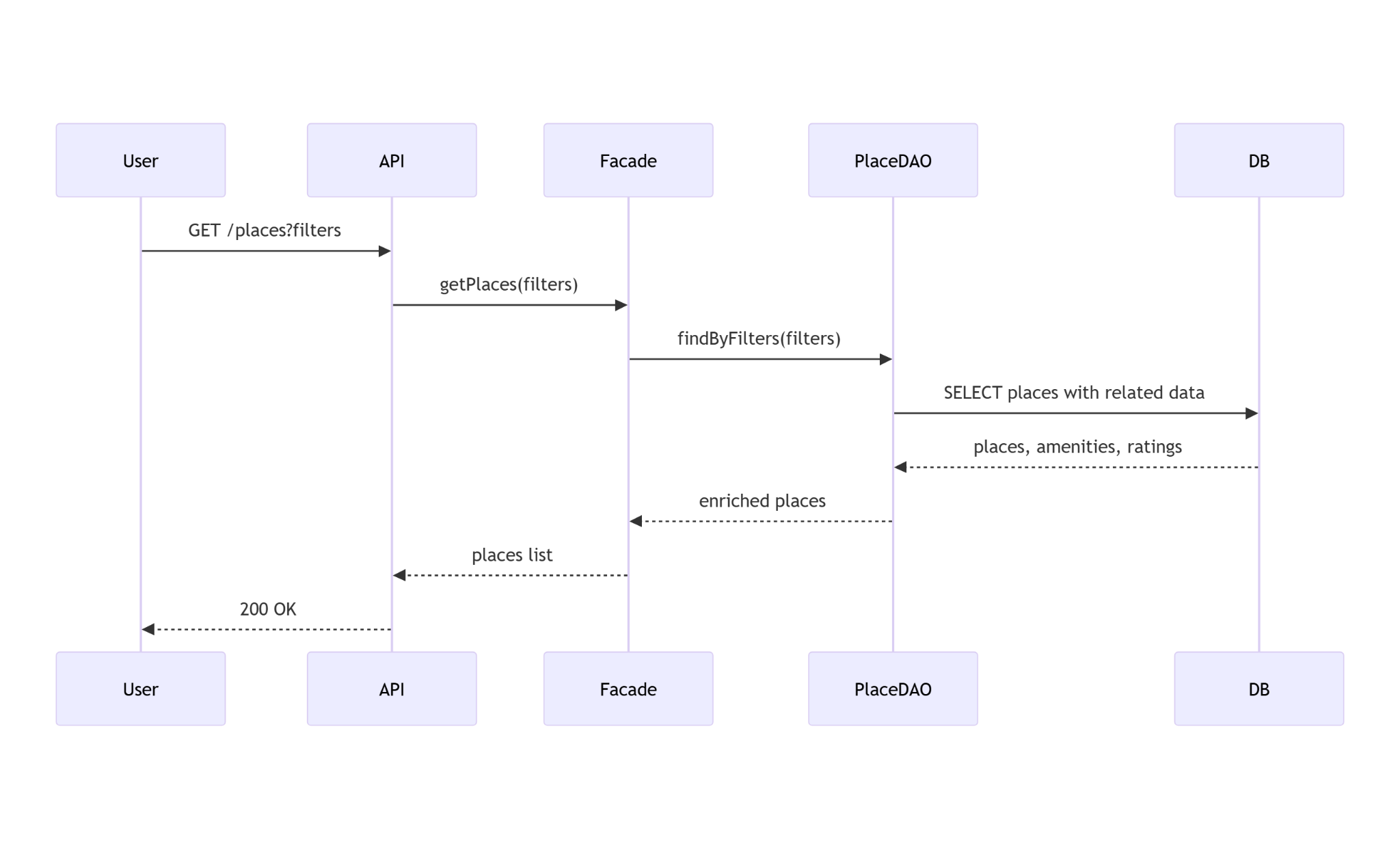


We return once again to the teleprompter analogy, where the user interacts with the API so that an account is stored into the app's database. As the request to create a user is issued, the API's facade redirects to the colloquial functions that allow the user to enter their credentials to be stored inside of the database to store for later. The data obtained is as per request of the app, typically the name, email and password for the account. Additional data is stored, such as the date of creation, other relevant information and the capacity to operate as an administrator. This process is appropriately separated by two options at prompt: to call for a returning user or to sign up as a new user to the application. Should the user be a new registry, the Facade drives the information into the user DAO for data registry. Upon a successful save, we return to the API once again, be it for the registry of another user or the fetching of information stored in the database. To appropriately register or fetch that information, the presentation layer ensures that the correct encrypted data is fed before anything is retrieved. Typically this occurs by way of passwords and authentication, which coincides between the user and the data stored. Once the innermost layers of the database query obtain the correct credentials, these IDs are stored into the navigating user's ID.

Should our user be an administrator, we can now proceed to our second diagram, which is the sequence of events for the creation of data to store a place.

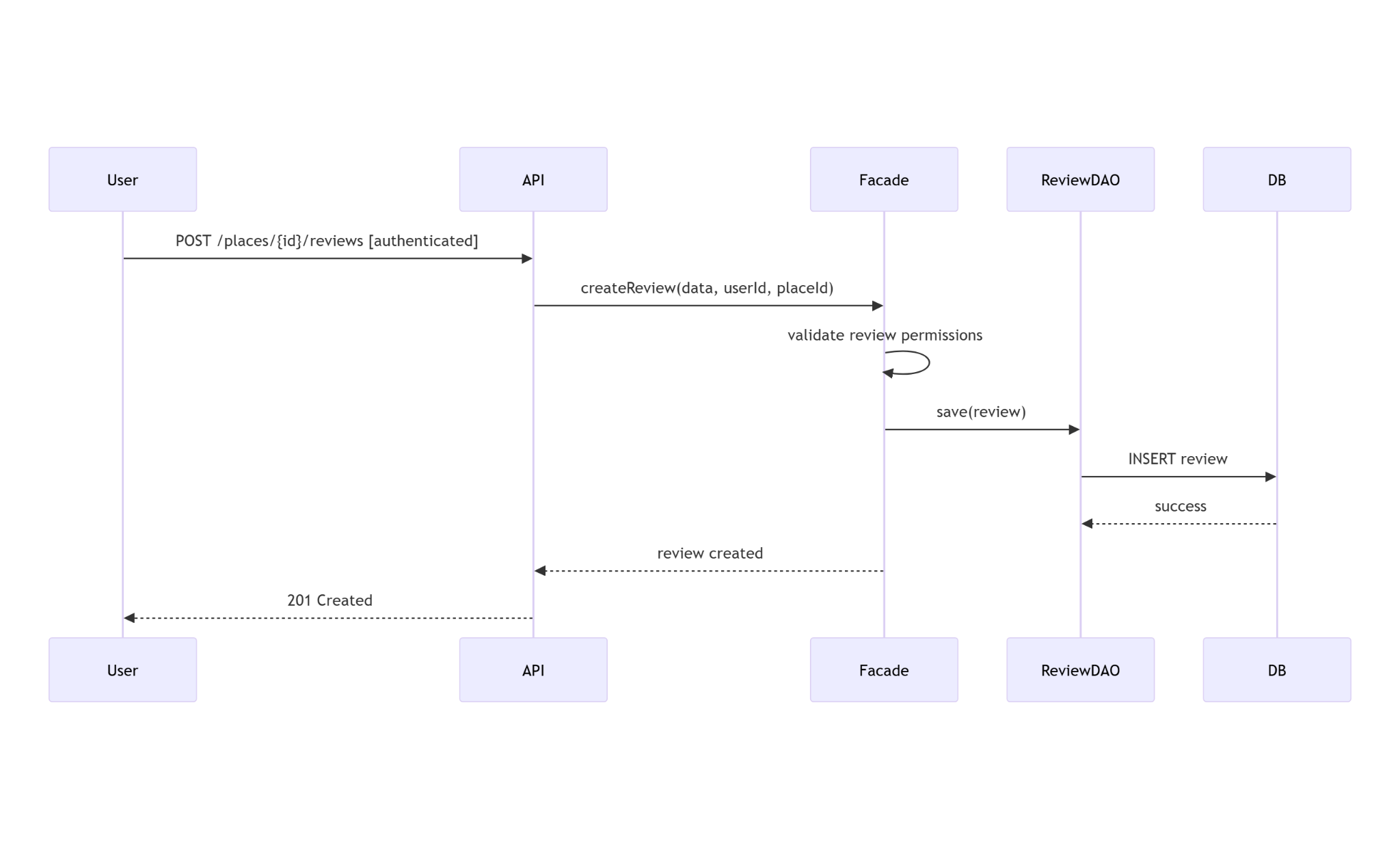


As stated first, the API is fed the request by the user admin to create a place and store in its database, for which the Facade redirects to the appropriate function that takes the information of the place and are stored, once again, as credentials that are tied to the user with its corresponding ID. As mentioned earlier, the composition of a place requires that more than one manageable field is entered. So the user is not just creating a place, but also creating amenities for it to store as part of the data tied to the place entity; all of it, again, stored in the same place ID. Note that amenities still exist as individual entities, which means that although it has a relation to the place to which it’s added, it contains its own ID as well, and has its own separate database, which is taken into consideration for the searching of places by the user. Upon a successful creation, the API returns the correct encryption code to return a value of success upon creation. Note, for instance, that creating places and amenities stand as their own individual objects, but for the successful creation of a place, the validation of both place and amenities must coincide with their respective IDs. This will ensure that both entities are linked appropriately but still existing within their respective database for security and clarity; more so than on search, but upon the addition and validation of new amenities. This is where the modulation of entities flourish; a user should not be restrained to access amenities by searching first through its places. Flexibility should be given for the customer to explore, but also for the administrator to effectively manage. Upon a successful creation, places can now be searched by potential customers and viewed in detail with all of its corresponding information on as described by our following diagram.



Upon searching, the option should appear for booking to be made from a particular date and time. This will require a set of fields, such as the time of encounter and afterwards the time of leaving. Having provided all the correct information for booking, a message should appear confirming that their stay was confirmed for the assigned time. And thus, the experience begins for them and for the application to do its work.

Ultimately, all that would remain is for our customers to enjoy their trip and, finally, leave a review for future customers to take into consideration. And thus, our final sequence for the complete experience.



When traversing the site, users should be allowed to revisit places and submit a review for future customers to see. The creation of these reviews are linked to the place and to the reviewer, so the review-ee will not be taken into account. This is an important interaction for the database, as the IDs that will be committed into the review databases are the ones linked to both the place and the reviewer, once again, highlighting the relation between both of these entities as described in our previous diagrams. As the customer searches for the place in which they stayed, the option to submit the review will appear upon their API. Having requested the option, the Facade directs the user to submit the review based on a commentary and a 0 to 5 star rating. Both fields should be considered important, as the database will require both in order to proceed with the submission. Finally, upon a successful submission, a message shall appear that says that the review has been submitted, which will be confirmed when researching that particular place and finding reviews upon the review window in the interface.