HBnB Technical Documentation

Complete System Architecture and Design

Document Version: 1.0 (Final)
Creation Date: October 5, 2025
Author: HBnB Development Team
Status: Complete and Validated

Table of Contents

---

1. Introduction

1.1 Document Purpose

This technical document serves as the complete blueprint for the HBnB (Holberton BnB) project, a property rental application similar to Airbnb. It provides a detailed reference for all implementation phases and offers a clear vision of the system architecture, data models, and interaction flows.

1.2 Project Scope

The HBnB project is a platform that allows users to:

- Create and manage user accounts (hosts and travelers)
- Publish and manage property listings for rent
- Search and filter accommodations based on various criteria
- Submit and view reviews on properties

- Manage amenities associated with places

1.3 Overall Architecture

The application follows a layered architecture with three distinct tiers:

- **Presentation Layer:** Manages user interfaces (REST API, Web, Mobile)
- **Business Logic Layer:** Business logic and data models
- **Persistence Layer:** Data access and storage

This separation ensures maintainability, scalability, and decoupling between components.

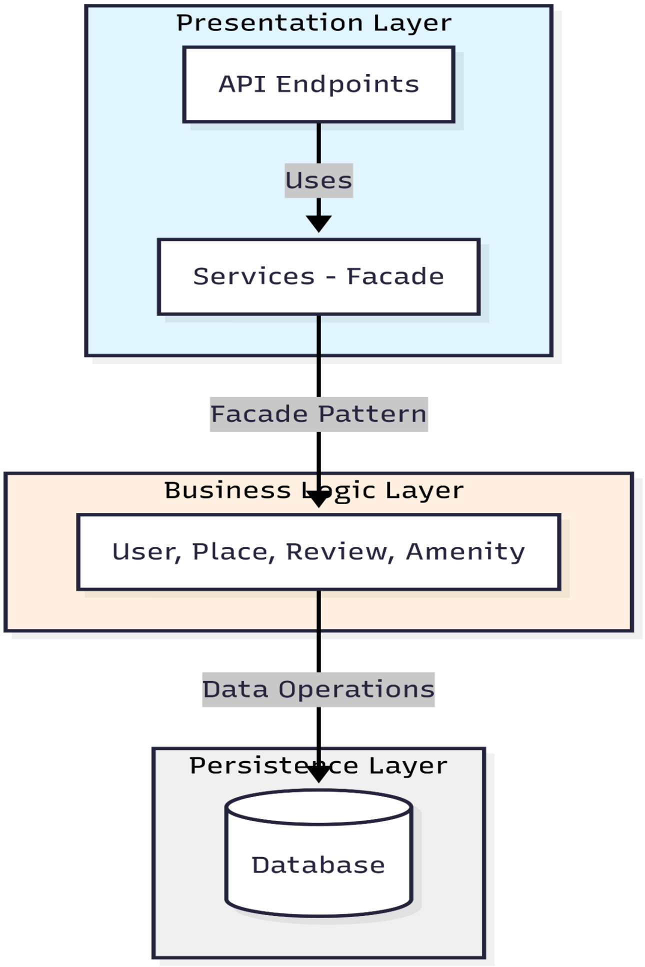
---

2. High-Level Architecture

2.1 Three-Layer Architecture Overview

The HBnB application is built using a three-layer architecture pattern that provides clear separation of concerns and facilitates independent development and testing of each layer.

DIAGRAM 1: High-Level Package Diagram



2.2 Layer Descriptions

2.2.1 Presentation Layer

Responsibilities:

- Handle incoming HTTP requests (REST API)
- User interfaces (Web and Mobile)
- Routing and endpoint control
- User input validation
- Response formatting (JSON, HTML)

Main Components:

- **API Endpoints:** Entry points for client applications
  - User endpoints: `/api/v1/users`
  - Place endpoints: `/api/v1/places`
  - Review endpoints: `/api/v1/reviews`
  - Amenity endpoints: `/api/v1/amenities`
- **Services - Facade:** Unified interface to Business Logic Layer

Communication Flow:

Client → API Endpoints → Facade (Services) → Business Logic Layer

Suggested Technologies: Flask/FastAPI (Python), Express.js (Node.js)

2.2.2 Facade Pattern

Role:
The Facade pattern serves as a unified interface between the presentation layer and business logic layer. It acts as a central orchestration point that simplifies interactions by hiding the underlying system's complexity.

Key Benefits:

- ✅ **Decoupling:** Presentation layer doesn't know business logic implementation details
- ✅ **Simplicity:** Single interface for all operations
- ✅ **Maintainability:** Internal changes without impacting presentation layer
- ✅ **Reusability:** Business code reusable by different interfaces (Web, Mobile, API)

Operation Example:

```
# Without Facade (tight coupling)
user = UserRepository.get(user_id)
```

```
place = PlaceRepository.add(place_data)
amenity1 = AmenityRepository.get(amenity_id1)
place.add_amenity(amenity1)

# With Facade (loose coupling)
facade.create_place(place_data)  # Handles everything internally
```

2.2.3 Business Logic Layer

Responsibilities:

- Implementation of business rules
- Domain entity management (User, Place, Review, Amenity)
- Business data validation
- Complex operation coordination
- Entity relationships management

Main Components:

- **Core Models:** Entity representation (User, Place, Review, Amenity)
- **Business Rules:** Validation logic and business processes
- **Domain Services:** Cross-cutting services (authentication, calculations)

Business Rule Examples:

- A user cannot review their own place (user_id ≠ place.owner_id)
- Review rating must be between 1 and 5
- A place must have at least one owner
- Email addresses must be unique
- Price per night must be greater than 0

2.2.4 Persistence Layer

Responsibilities:

- Data access and storage
- CRUD operations (Create, Read, Update, Delete)
- Transaction management
- Object-relational mapping (ORM)
- Database connection pooling
- Query optimization

Main Components:

- **Repositories:** Data access abstraction

  - UserRepository: User data operations
  - PlaceRepository: Place data operations
  - ReviewRepository: Review data operations
  - AmenityRepository: Amenity data operations
- **Database:** Relational database (PostgreSQL, MySQL)
- **Connection Management:** Database connection handling

Repository Pattern Benefits:

- Abstraction of data access logic
- Centralized query management
- Easy testing with mock repositories
- Flexibility to change database technology

Communication Flow:

```
Business Logic → Repository → Database
                ↓
        SQL/ORM Queries
```

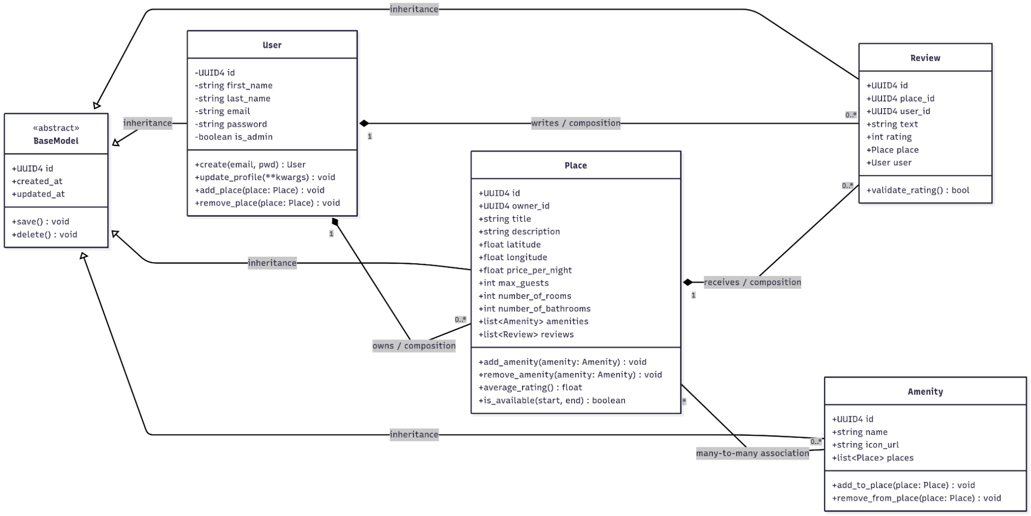
---

3. Business Logic Layer - Detailed Class Diagram

3.1 Entity Model Overview

The Business Logic Layer consists of four main entities and one abstract base class that provides common functionality. These entities represent the core domain objects of the HBnB application.

DIAGRAM 2: Detailed Class Diagram



3.2 Detailed Entity Descriptions

3.2.1 BaseModel (Abstract Class)

Description: Base abstract class that provides common functionality for all entities

Attributes:

- `id: UUID4` - Unique identifier
- `created_at: DateTime` - Entity creation timestamp
- `updated_at: DateTime` - Last modification timestamp

Methods:

- `save(): void` - Persist entity to database
- `delete(): void` - Delete entity from database

Purpose: Provides timestamp tracking and common persistence methods for all domain entities

3.2.2 User

Description: Represents a platform user (host or traveler)

Attributes:

- `id: UUID4` - Unique identifier (inherited from BaseModel)
- `first_name: string` - User's first name
- `last_name: string` - User's last name
- `email: string` - Email address (unique, used for authentication)
- `password: string` - Hashed password (bcrypt/argon2)
- `is_admin: boolean` - Administrator privileges flag
- `created_at: DateTime` - Account creation date (inherited)
- `updated_at: DateTime` - Last modification date (inherited)

Methods:

- `create(email, pwd): User` - Create a new user account
- `update_profile(**kwargs): void` - Update profile information
- `add_place(place: Place): void` - Add a place owned by this user
- `remove_place(place: Place): void` - Remove a place from user's listings

Relationships:

- 1 User → 0..* Places (owns/composition): A user can own multiple places
- 1 User → 0..* Reviews (writes/composition): A user can write multiple reviews

Business Rules:

- Email must be unique in the system
- Password minimum 8 characters
- Email format validation (RFC 5322)
- First name and last name are required
- Cannot delete user if they have active bookings

Validation:

- Email: valid format, unique
- Password: minimum 8 chars, contains uppercase, lowercase, number
- Names: non-empty strings

3.2.3 Place

Description: Represents a property available for rent

Attributes:

- `id: UUID4` - Unique identifier (inherited from BaseModel)
- `owner_id: UUID4` - Reference to owner user
- `title: string` - Listing title
- `description: string` - Detailed place description
- `latitude: float` - GPS latitude coordinate
- `longitude: float` - GPS longitude coordinate
- `price_per_night: float` - Nightly rental price
- `max_guests: int` - Maximum number of guests
- `number_of_rooms: int` - Number of bedrooms
- `number_of_bathrooms: int` - Number of bathrooms
- `amenities: list<Amenity>` - List of available amenities
- `reviews: list<Review>` - List of reviews for this place
- `created_at: DateTime` - Listing creation date (inherited)
- `updated_at: DateTime` - Last modification date (inherited)

Methods:

- `add_amenity(amenity: Amenity): void` - Add amenity to place
- `remove_amenity(amenity: Amenity): void` - Remove amenity from place

- `average_rating(): float` - Calculate average rating from reviews
- `is_available(start, end): boolean` - Check availability for date range

Relationships:

- 1 User → 0..* Places (owns): Each place belongs to one owner
- 1 Place → 0..* Reviews (has/composition): A place can have multiple reviews
- Place ↔ Amenity (many-to-many): A place has multiple amenities, an amenity belongs to multiple places

Business Rules:

- Price per night must be > 0
- Valid GPS coordinates: -90 ≤ latitude ≤ 90, -180 ≤ longitude ≤ 180
- Number of rooms/bathrooms must be ≥ 0
- Max guests must be > 0
- Title is required (non-empty)
- Owner must exist (foreign key constraint)

Validation:

- price_per_night: > 0, float
- latitude: -90 to 90
- longitude: -180 to 180
- max_guests: > 0, integer
- title: non-empty string
- description: optional string

3.2.4 Review

Description: Represents a review left by a user on a place

Attributes:

- `id: UUID4` - Unique identifier (inherited from BaseModel)
- `place_id: UUID4` - Reference to reviewed place
- `user_id: UUID4` - Reference to review author
- `text: string` - Review text/comment
- `rating: int` - Rating from 1 to 5 stars
- `place: Place` - Reference to Place object
- `user: User` - Reference to User object
- `created_at: DateTime` - Review creation date (inherited)
- `updated_at: DateTime` - Last modification date (inherited)

Methods:

- `validate_rating(): bool` - Validate rating is between 1 and 5

Relationships:

- 1 User → 0..* Reviews (writes): A user can write multiple reviews
- 1 Place → 0..* Reviews (receives/composition): A place can receive multiple reviews

Business Rules:

- **CRITICAL:** A user CANNOT review their own place (user_id ≠ place.owner_id)
- Rating mandatory and must be between 1 and 5 (inclusive)
- One review per user per place (unique constraint on user_id + place_id)
- Text comment is optional but recommended
- Review can only be created after a completed booking (optional rule)

Validation:

- rating: integer between 1 and 5 (inclusive)
- user_id: must exist and not be place owner
- place_id: must exist
- text: optional string, max 1000 characters
- ownership: user_id != place.owner_id

3.2.5 Amenity

Description: Represents equipment or service available at a place

Attributes:

- `id: UUID4` - Unique identifier (inherited from BaseModel)
- `name: string` - Amenity name (e.g., "WiFi", "Pool", "Parking")
- `icon_url: string` - URL to icon image
- `places: list<Place>` - List of places with this amenity
- `created_at: DateTime` - Creation date (inherited)
- `updated_at: DateTime` - Last modification date (inherited)

Methods:

- `add_to_place(place: Place): void` - Add this amenity to a place
- `remove_from_place(place: Place): void` - Remove this amenity from a place

Relationships:

- Place ↔ Amenity (many-to-many): A place has multiple amenities, an amenity belongs to multiple places

Business Rules:

- Amenity name must be unique
- Name is required (non-empty)
- Icon URL should be valid URL format
- Predefined list of standard amenities (WiFi, Pool, Parking, Kitchen, etc.)

Validation:

- name: unique, non-empty string, max 50 characters
- icon_url: valid URL format

Common Amenities:

- WiFi
- Swimming Pool
- Parking
- Kitchen
- Air Conditioning
- Heating
- TV
- Washing Machine
- Workspace

3.3 Relationships and Cardinalities Summary

Relationship	Type	Cardinality	Implementation	Description
User → Place	One-to-Many	1:0..*	Foreign Key (owner_id)	A user owns multiple places
User → Review	One-to-Many	1:0..*	Foreign Key (user_id)	A user writes multiple reviews
Place → Review	One-to-Many	1:0..*	Foreign Key (place_id)	A place receives multiple reviews
Place ↔ Amenity	Many-to-Many	*:*	Join Table (place_amenities)	Bidirectional association
BaseModel → All	Inheritance	-	Class Inheritance	Provides common attributes/methods

4. API Interaction Flow - Sequence Diagrams

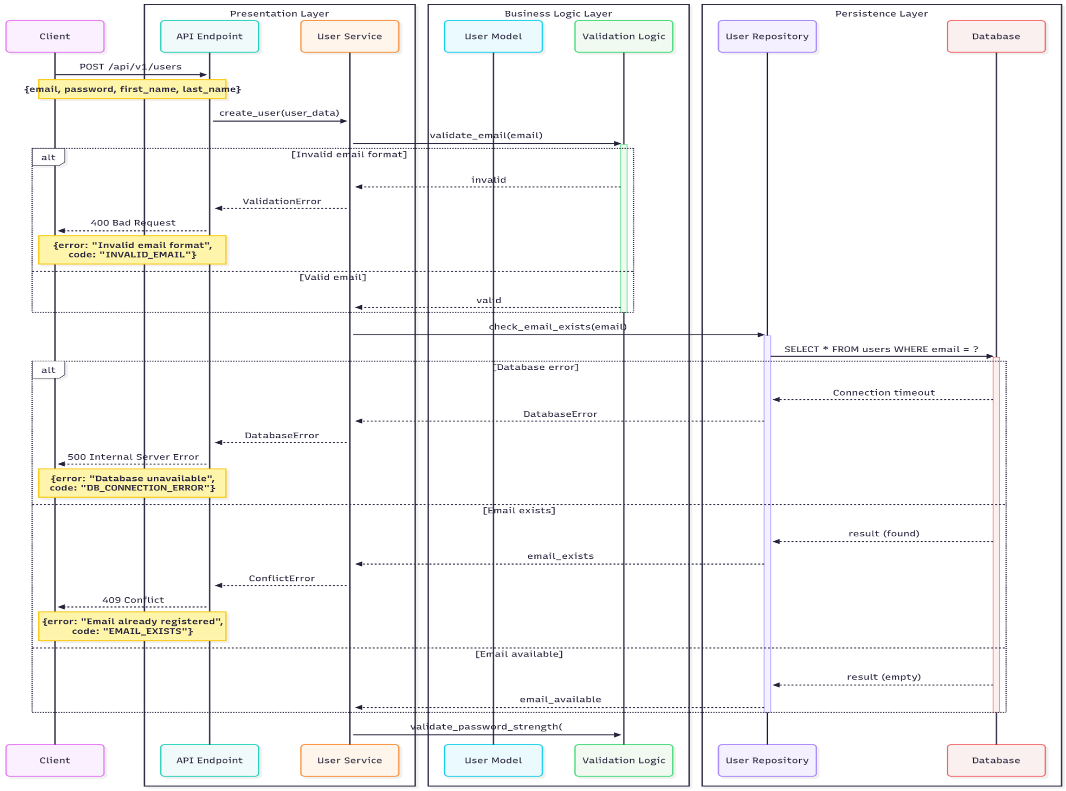
This section presents detailed sequence diagrams for the four main API operations, including complete HTTP error handling and validation flows.

4.1 User Registration

4.1.1 Overview

The User Registration flow demonstrates how a new user account is created in the system, including validation and secure password hashing.

DIAGRAM 3: User Registration Sequence Diagram



4.1.2 Flow Description

Objective: Create a new user account in the system
API Endpoint: `POST /api/v1/users`

Request Body:

```
{
  "email": "john.doe@example.com",
  "password": "SecurePass123!",
  "first_name": "John",
  "last_name": "Doe"
}
```

HTTP Status Codes:

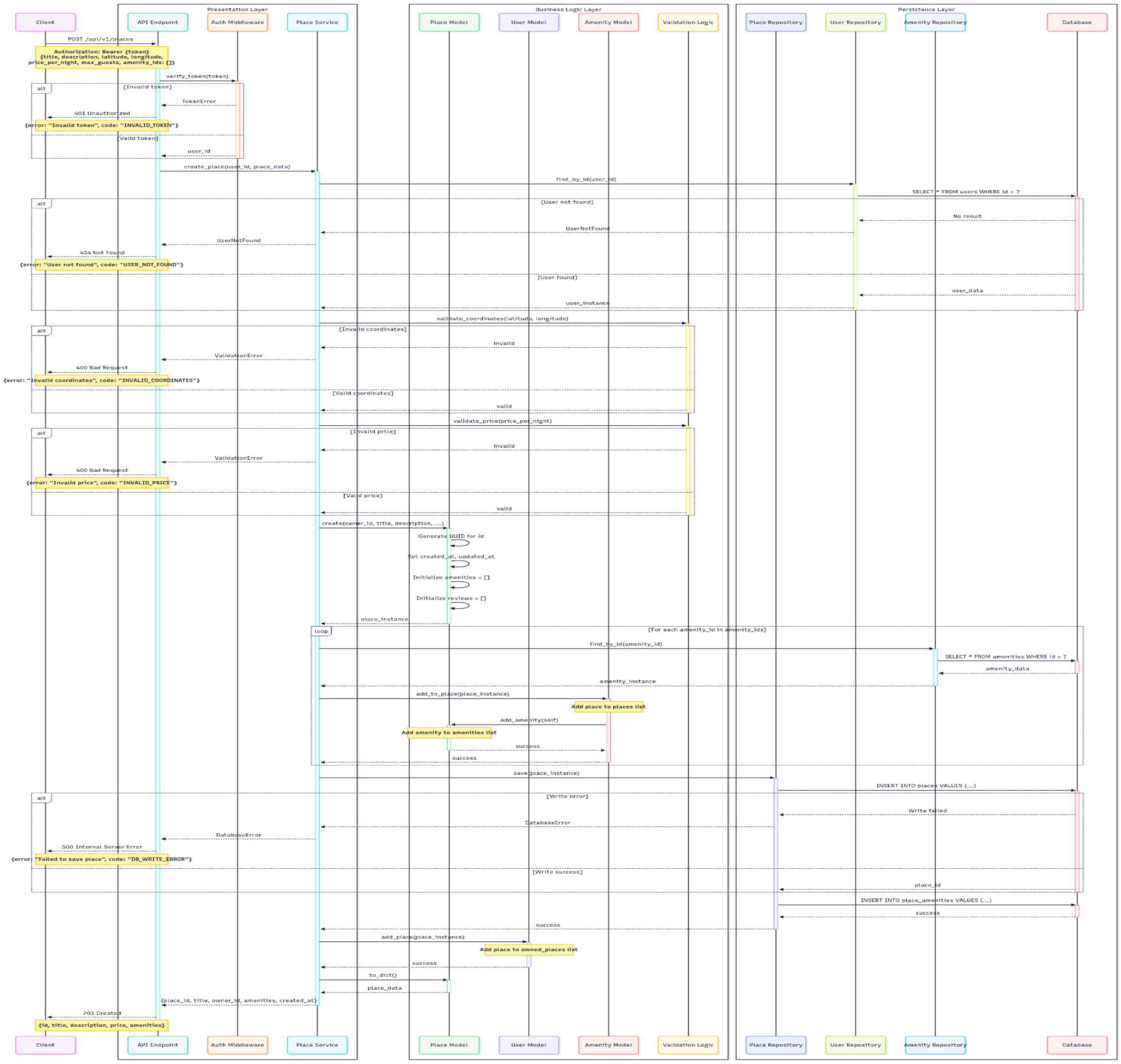
- `201 Created`: User successfully created
- `400 Bad Request`: Invalid request data or validation failure
- `409 Conflict`: Email already exists
- `500 Internal Server Error`: Database or server error

4.2 Place Registration

4.2.1 Overview

The Place Registration flow demonstrates how a new property listing is created in the system, including multi-step validation of the owner, amenities, and business rules.

DIAGRAM 4: Place Registration Sequence Diagram


Presentation Layer
Business Logic Layer
Persistence Layer
Client
API Endpoint
Auth Middleware
Place Service
Place Model
User Model
Amenity Model
Validation Logic
Place Repository
User Repository
Amenity Repository
Database
POST /api/v1/places
Authorization: Bearer {token}
{title, description, latitude, longitude,
price_per_night, max_guests, amenity_ids: []}
verify_token(token)
alt
[Invalid token]
TokenError
401 Unauthorized
{error: "Invalid token", code: "INVALID_TOKEN"}
[Valid token]
user_id
create_place(user_id, place_data)
find_by_id(user_id)
SELECT * FROM users WHERE id = ?
alt
[User not found]
No result
UserNotFound
UserNotFound
404 Not Found
{error: "User not found", code: "USER_NOT_FOUND"}
[User found]
user_data
user_instance
validate_coordinates(latitude, longitude)
alt
[Invalid coordinates]
invalid
ValidationError
400 Bad Request
{error: "Invalid coordinates", code: "INVALID_COORDINATES"}
[Valid coordinates]
valid
validate_price(price_per_night)
alt
[Invalid price]
invalid
ValidationError
400 Bad Request
{error: "Invalid price", code: "INVALID_PRICE"}
[Valid price]
valid
create(owner_id, title, description, ...)
Generate UUID for id
Set created_at, updated_at
Initialize amenities = []
Initialize reviews = []
place_instance
loop
[For each amenity_id in amenity_ids]
find_by_id(amenity_id)
SELECT * FROM amenities WHERE id = ?
amenity_data
amenity_instance
add_to_place(place_instance)
Add place to places list
add_amenity(self)
Add amenity to amenities list
success
success
save(place_instance)
INSERT INTO places VALUES (...)
alt
[Write error]
Write failed
DatabaseError
DatabaseError
500 Internal Server Error
{error: "Failed to save place", code: "DB_WRITE_ERROR"}
[Write success]
place_id
INSERT INTO place_amenities VALUES (...)
success
success
add_place(place_instance)
Add place to owned_places list
success
to_dict()
place_data
{place_id, title, owner_id, amenities, created_at}
201 Created
{id, title, description, price, amenities}

4.2.2 Flow Description

Objective: Create a new place listing in the system
API Endpoint: `POST /api/v1/places`

Request Body:

```
{
  "title": "Cozy Downtown Apartment",
  "description": "Beautiful 2-bedroom apartment in the heart of the city",
  "price_per_night": 85.00,
  "latitude": 48.8566,
  "longitude": 2.3522,
  "max_guests": 4,
  "number_of_rooms": 2,
  "number_of_bathrooms": 1,
  "owner_id": "user-uuid-123",
  "amenities": ["amenity-uuid-1", "amenity-uuid-2"]
}
```

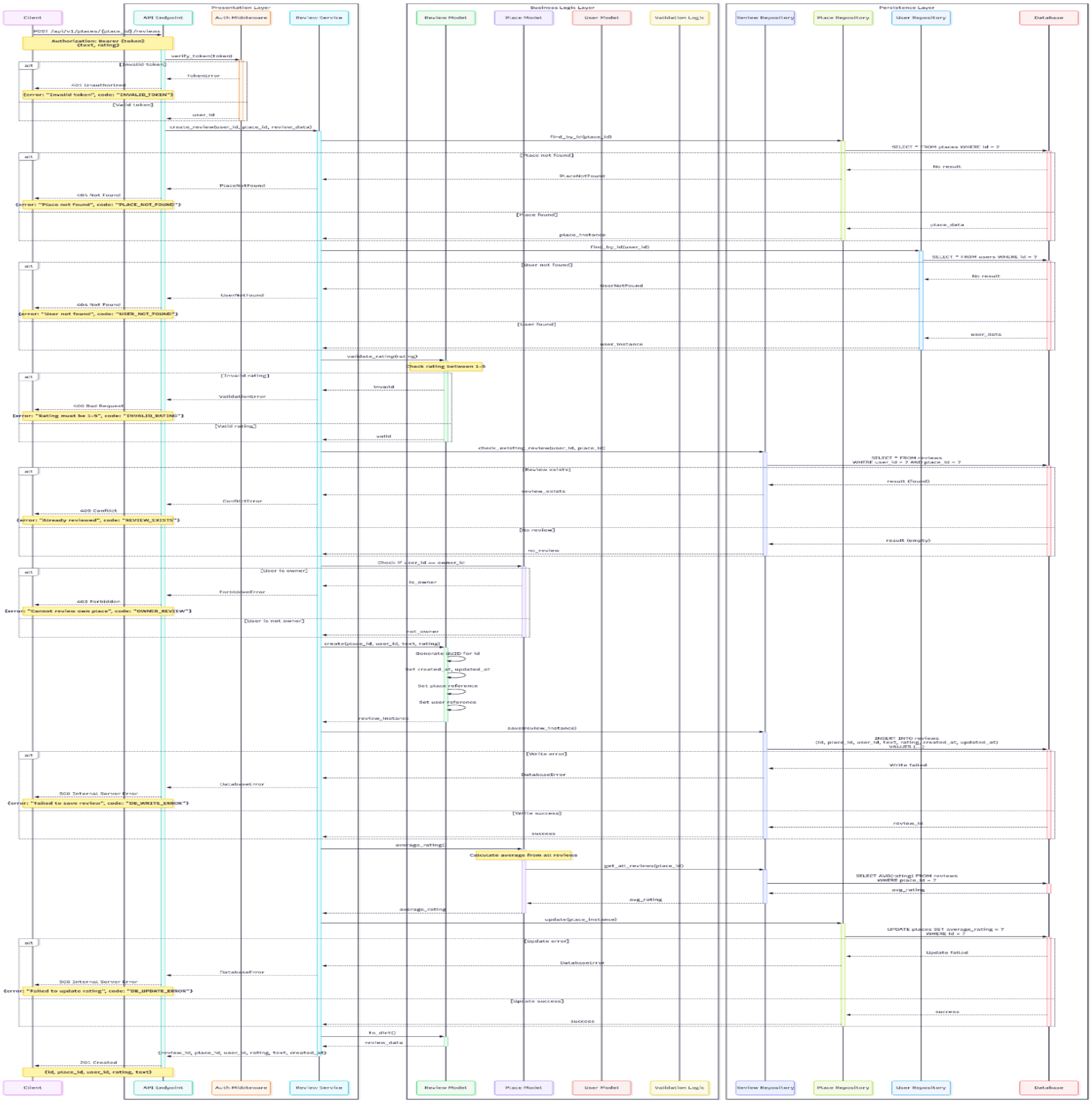
HTTP Status Codes:

- `201 Created`: Place successfully created
- `400 Bad Request`: Invalid request data, invalid amenity, or business rule violation
- `401 Unauthorized`: Invalid or missing authentication token
- `404 Not Found`: Owner user not found
- `500 Internal Server Error`: Database or server error

4.3 Review Submission

4.3.1 Overview

The Review Submission flow shows how a user can submit a review for a place, with a critical business rule that prevents owners from reviewing their own properties.


Presentation Layer
Business Logic Layer
Persistence Layer
Client
API Endpoint
Auth Middleware
Review Service
Review Model
Place Model
User Model
Validation Logic
Review Repository
Place Repository
User Repository
Database
POST /api/v1/places/{place_id}/reviews
Authorization: Bearer {token}
{text, rating}
verify_token(token)
alt
[Invalid token]
TokenError
401 Unauthorized
{error: "Invalid token", code: "INVALID_TOKEN"}
[Valid token]
user_id
create_review(user_id, place_id, review_data)
find_by_id(place_id)
SELECT * FROM places WHERE id = ?
alt
[Place not found]
No result
PlaceNotFound
PlaceNotFound
404 Not Found
{error: "Place not found", code: "PLACE_NOT_FOUND"}
[Place found]
place_data
place_instance
find_by_id(user_id)
SELECT * FROM users WHERE id = ?
alt
[User not found]
No result
UserNotFound
UserNotFound
404 Not Found
{error: "User not found", code: "USER_NOT_FOUND"}
[User found]
user_data
user_instance
validate_rating(rating)
Check rating between 1-5
alt
[Invalid rating]
invalid
ValidationError
400 Bad Request
{error: "Rating must be 1-5", code: "INVALID_RATING"}
[Valid rating]
valid
check_existing_review(user_id, place_id)
SELECT * FROM reviews
WHERE user_id = ? AND place_id = ?
alt
[Review exists]
result (found)
review_exists
ConflictError
409 Conflict
{error: "Already reviewed", code: "REVIEW_EXISTS"}
[No review]
result (empty)
no_review
Check if user_id == owner_id
alt
[User is owner]
is_owner
ForbiddenError
403 Forbidden
{error: "Cannot review own place", code: "OWNER_REVIEW"}
[User is not owner]
not_owner
create(place_id, user_id, text, rating)
Generate UUID for id
Set created_at, updated_at
Set place reference
Set user reference
review_instance
save(review_instance)
INSERT INTO reviews
(id, place_id, user_id, text, rating, created_at, updated_at)
VALUES (...)
alt
[Write error]
Write failed
DatabaseError
DatabaseError
500 Internal Server Error
{error: "Failed to save review", code: "DB_WRITE_ERROR"}
[Write success]
review_id
success
average_rating()
Calculate average from all reviews
get_all_reviews(place_id)
SELECT AVG(rating) FROM reviews
WHERE place_id = ?
avg_rating
avg_rating
average_rating
update(place_instance)
UPDATE places SET average_rating = ?
WHERE id = ?
alt
[Update error]
Update failed
DatabaseError
DatabaseError
500 Internal Server Error
{error: "Failed to update rating", code: "DB_UPDATE_ERROR"}
[Update success]
success
success
to_dict()
review_data
{review_id, place_id, user_id, rating, text, created_at}
201 Created
{id, place_id, user_id, rating, text}

4.3.2 Flow Description

Objective: Allow a user to submit a review on a place
API Endpoint: `POST /api/v1/reviews`

Request Body:

```
{
  "place_id": "place-uuid-456",
  "user_id": "user-uuid-789",
  "rating": 5,
  "text": "Excellent stay! The place was clean, comfortable, and perfectly located."
}
```

Critical Business Rules:

- **Owner Restriction:** User cannot review their own place (prevents bias)
- **Rating Range:** Rating must be 1-5 (data integrity)
- **Uniqueness:** One review per user per place (prevents spam)
- **Cascade Update:** Average rating automatically recalculated

HTTP Status Codes:

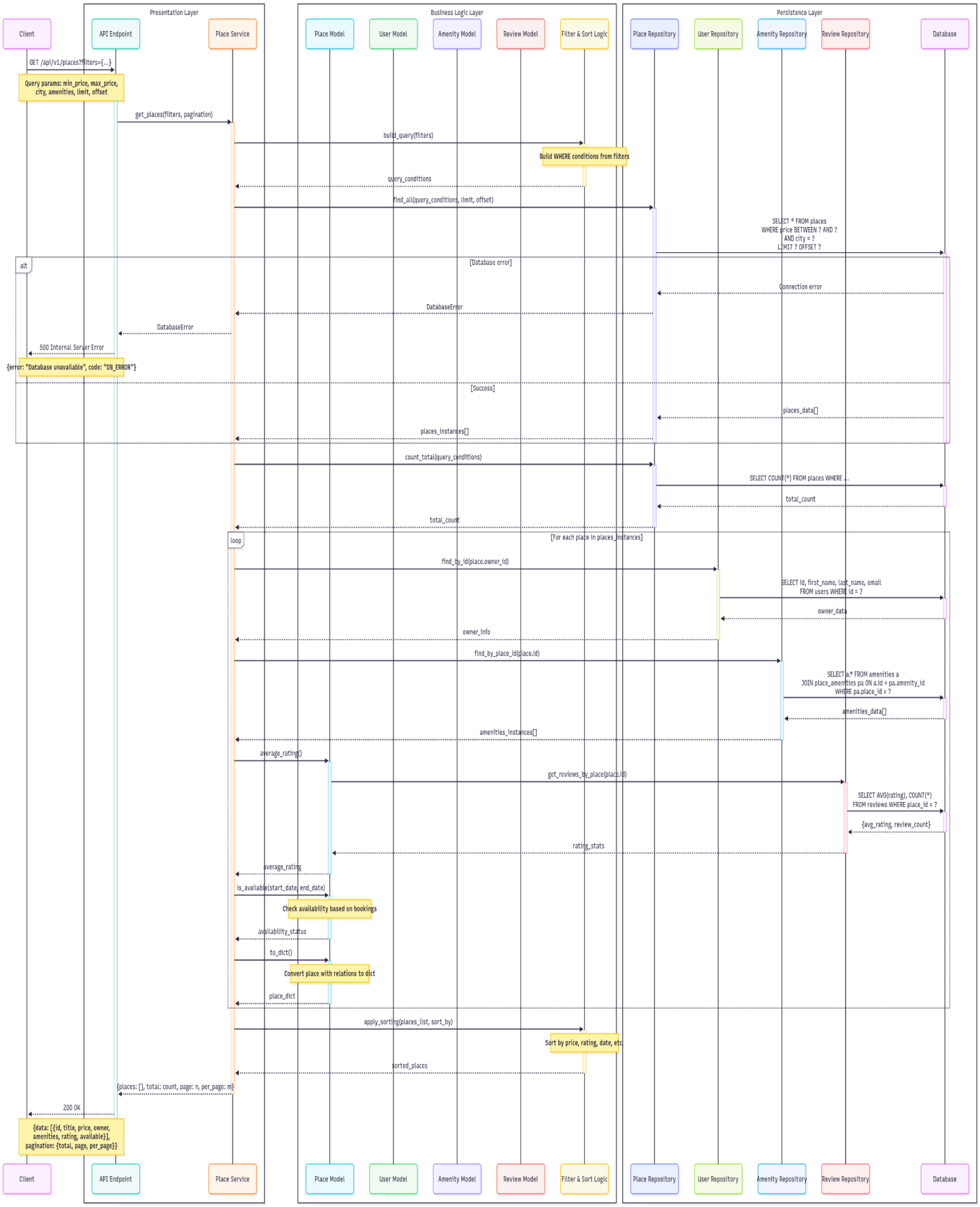
- `201 Created`: Review successfully created
- `400 Bad Request`: Invalid data, invalid rating, or owner reviewing own place
- `401 Unauthorized`: Invalid or missing authentication token
- `404 Not Found`: Place or user not found
- `409 Conflict`: Review already exists for this user-place combination
- `500 Internal Server Error`: Database or server error

4.4 Fetching Places List

4.4.1 Overview

The Fetching Places List flow demonstrates how clients can retrieve a filtered and paginated list of available places, with support for various search criteria.

DIAGRAM 6: Fetching Places List Sequence Diagram


Presentation Layer
Business Logic Layer
Persistence Layer
Client
API Endpoint
Place Service
Place Model
User Model
Amenity Model
Review Model
Filter & Sort Logic
Place Repository
User Repository
Amenity Repository
Review Repository
Database
GET /api/v1/places?filters={...}
Query params: min_price, max_price, city, amenities, limit, offset
get_places(filters, pagination)
build_query(filters)
Build WHERE conditions from filters
query_conditions
find_all(query_conditions, limit, offset)
SELECT * FROM places WHERE price BETWEEN ? AND ? AND city = ? LIMIT ? OFFSET ?
alt
[Database error]
Connection error
DatabaseError
DatabaseError
500 Internal Server Error
{error: "Database unavailable", code: "DB_ERROR"}
[Success]
places_data[]
places_instances[]
count_total(query_conditions)
SELECT COUNT(*) FROM places WHERE ...
total_count
total_count
loop
[For each place in places_instances]
find_by_id(place.owner_id)
SELECT id, first_name, last_name, email FROM users WHERE id = ?
owner_data
owner_info
find_by_place_id(place.id)
SELECT a.* FROM amenities a JOIN place_amenities pa ON a.id = pa.amenity_id WHERE pa.place_id = ?
amenities_data[]
amenities_instances[]
average_rating()
get_reviews_by_place(place.id)
SELECT AVG(rating), COUNT(*) FROM reviews WHERE place_id = ?
{avg_rating, review_count}
rating_stats
average_rating
is_available(start_date, end_date)
Check availability based on bookings
availability_status
to_dict()
Convert place with relations to dict
place_dict
apply_sorting(places_list, sort_by)
Sort by price, rating, date, etc
sorted_places
{places: [], total: count, page: n, per_page: m}
200 OK
{data: [{id, title, price, owner, amenities, rating, available}], pagination: {total, page, per_page}}
Client
API Endpoint
Place Service
Place Model
User Model
Amenity Model
Review Model
Filter & Sort Logic
Place Repository
User Repository
Amenity Repository
Review Repository
Database

4.4.2 Flow Description

Objective: Retrieve a list of available places with filters and pagination
API Endpoint: `GET /api/v1/places`

Query Parameters:

```
GET
/api/v1/places?city=Paris&min_price=50&max_price=200&amenities=wifi,pool&limit=20&offset=0
```

Response Example:

```
{
  "data": [
    {
      "id": "place-1",
      "title": "Marais Apartment",
      "description": "Charming 2-bedroom...",
      "price_per_night": 85.00,
      "city": "Paris",
      "owner": {
        "id": "user-123",
        "first_name": "John",
        "last_name": "Doe"
      },
      "amenities": [
        {"id": "am-1", "name": "WiFi"},
        {"id": "am-2", "name": "Pool"}
      ],
      "average_rating": 4.5,
      "review_count": 23
    }
  ],
  "pagination": {
    "limit": 20,
    "offset": 0,
    "total_count": 45,
    "page": 1,
    "total_pages": 3
  }
}
```

HTTP Status Codes:

- `200 OK`: Request successful (even if results are empty)
- `400 Bad Request`: Invalid query parameters
- `500 Internal Server Error`: Database or server error

Filter Options:

- `city`: Filter by city name
- `min_price` / `max_price`: Price range filter
- `amenities`: Filter by one or more amenities
- `limit` / `offset`: Pagination controls
- `sort_by`: Sort by price, rating, date

Query Optimization Strategies:

- **Indexing:** Indexes on city, price_per_night, created_at
- **Eager Loading:** Load related data efficiently
- **Query Batching:** Batch queries for multiple places
- **Caching:** Cache frequently accessed filters
- **Pagination:** Limit results to avoid memory issues

---

5. Design Decisions

5.1 Layered Architecture

Decision: Use 3-layer architecture (Presentation, Business Logic, Persistence)

Justification:

- ✅ **Separation of Concerns:** Each layer has a clear, distinct responsibility
- ✅ **Maintainability:** Changes in one layer have minimal impact on others
- ✅ **Testability:** Each layer can be tested independently with mocking
- ✅ **Scalability:** Layers can be scaled independently or deployed as microservices

Implementation Approach:

- Clear interfaces between layers
- No direct database access from presentation layer
- Business logic isolated from HTTP concerns

5.2 Facade Pattern

Decision: Use Facade pattern between Presentation and Business Logic layers

Justification:

- ✅ **Unified Interface:** Single entry point for all business operations
- ✅ **Decoupling:** Presentation layer doesn't know business logic details
- ✅ **Simplicity:** Reduces complexity of client code
- ✅ **Orchestration:** Handles complex multi-step operations

5.3 Repository Pattern

Decision: Use Repository pattern for data access layer

Justification:

- ✅ **Abstraction:** Hides database implementation details
- ✅ **Centralization:** All data access logic in one place per entity
- ✅ **Testability:** Easy to mock for unit tests
- ✅ **Flexibility:** Can change database technology without affecting business logic

Standard Repository Interface:

```
class BaseRepository:
    def get(self, id: UUID) -> Optional[Entity]
    def get_all(self) -> List[Entity]
    def find_by(self, **criteria) -> List[Entity]
    def add(self, entity: Entity) -> Entity
    def update(self, entity: Entity) -> Entity
    def delete(self, id: UUID) -> bool
```

5.4 Cascade Validation

Decision: Validate all dependencies before creating/updating entities

Justification:

- ✅ **Data Integrity:** Ensures referential integrity
- ✅ **User Experience:** Returns all errors at once (no partial failures)
- ✅ **Performance:** Avoids expensive rollbacks
- ✅ **Atomic Operations:** All-or-nothing approach

Validation Sequence:

1. **Format Validation:** Data types, required fields, string lengths
2. **Business Rule Validation:** Domain-specific rules
3. **Reference Validation:** Check existence of related entities
4. **Uniqueness Validation:** Check for duplicates
5. **Final Creation/Update:** Only if all validations pass

5.5 Many-to-Many Relationship (Place ↔ Amenity)

Decision: Use join table `place_amenities` for many-to-many relationship

Justification:

- ✅ **Flexibility:** A place can have multiple amenities, an amenity in multiple places
- ✅ **Normalization:** Avoids data duplication
- ✅ **Performance:** Optimized queries with proper indexing

Implementation:

```
CREATE TABLE place_amenities (
    place_id UUID NOT NULL,
    amenity_id UUID NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (place_id, amenity_id),
    FOREIGN KEY (place_id) REFERENCES places(id) ON DELETE CASCADE,
    FOREIGN KEY (amenity_id) REFERENCES amenities(id) ON DELETE CASCADE
);
```

5.6 Timestamp Management

Decision: Add `created_at` and `updated_at` to all entities

Justification:

- ✅ **Audit Trail:** Track when records are created and modified
- ✅ **Debugging:** Essential for troubleshooting
- ✅ **Business Intelligence:** Analyze creation patterns
- ✅ **Compliance:** Required for GDPR and other regulations

5.7 HTTP Status Codes

Decision: Use standard HTTP status codes consistently

Status Code Mapping:

Code	Name	Use Case
200	OK	Successful GET request
201	Created	Successful POST (resource created)
204	No Content	Successful DELETE
400	Bad Request	Invalid data or business rule violation
401	Unauthorized	Missing or invalid authentication
403	Forbidden	Insufficient permissions
404	Not Found	Resource doesn't exist
409	Conflict	Resource conflict (duplicate)
500	Internal Server Error	Unexpected server error

Error Response Format:

```
{
  "error": "Human-readable error message",
  "code": "MACHINE_READABLE_ERROR_CODE",
  "details": {
    "field": "specific_field",
    "reason": "Additional context"
  }
}
```

5.8 Authentication Strategy

Decision: Use JWT (JSON Web Tokens) for stateless authentication

Justification:

- ✅ **Stateless:** No server-side session storage required
- ✅ **Scalable:** Works well with load balancers and microservices
- ✅ **Standard:** Industry-standard approach
- ✅ **Flexible:** Can include custom claims

Token Structure:

```
{
  "user_id": "uuid-123",
  "email": "user@example.com",
  "is_admin": false,
  "exp": 1696512000
}
```

---

6. Conclusion

6.1 Summary

This technical documentation presents the complete architecture and design of the HBnB project, including:

- ✅ **High-Level Architecture:** Three-layer architecture with clear separation
- ✅ **Facade Pattern:** Unified interface for simplified interactions
- ✅ **Detailed Class Diagram:** Complete entity models with relationships
- ✅ **Sequence Diagrams:** Four detailed API flows with error handling
- ✅ **Design Decisions:** Justified architectural choices

6.2 Implementation Roadmap

Phase 1: Foundation (Weeks 1-2)

- Environment setup (Python 3.10+, Flask/FastAPI, PostgreSQL)
- Project structure and configuration
- Database schema creation
- Basic models implementation

Phase 2: Persistence Layer (Weeks 3-4)

- Repository implementations
- Database connection pooling
- Transaction management
- Unit tests for repositories

Phase 3: Business Logic Layer (Weeks 5-6)

- Entity model implementations
- Business rule validation
- Facade implementation
- Unit tests for business logic

Phase 4: Presentation Layer (Weeks 7-8)

- API endpoint implementations
- Request/response validation
- Authentication middleware
- Integration tests

Phase 5: Advanced Features (Weeks 9-10)

- Search and filtering optimization
- Caching layer (Redis)
- Rate limiting
- API documentation (Swagger/OpenAPI)

Phase 6: Security & Performance (Weeks 11-12)

- Security audit
- Performance optimization
- Load testing
- Production deployment preparation

6.3 Technology Stack Recommendations

Backend:

- **Language:** Python 3.10+
- **Framework:** Flask or FastAPI
- **ORM:** SQLAlchemy 2.0
- **Database:** PostgreSQL 14+
- **Cache:** Redis 7+

Authentication:

- **JWT:** PyJWT
- **Password Hashing:** bcrypt or argon2

API Documentation:

- **OpenAPI 3.0:** Swagger UI or ReDoc

Development Tools:

- **Code Quality:** pylint, black, isort
- **Type Checking:** mypy
- **Testing:** pytest with pytest-cov
- **CI/CD:** GitHub Actions or GitLab CI

Deployment:

- **Containerization:** Docker + docker-compose
- **Monitoring:** Prometheus + Grafana
- **Logging:** ELK Stack

6.4 Key Architectural Principles

SOLID Principles Applied:

- **Single Responsibility:** Each class/layer has one clear purpose
- **Open/Closed:** Open for extension, closed for modification
- **Liskov Substitution:** BaseModel inheritance properly implemented
- **Interface Segregation:** Clear interfaces between layers
- **Dependency Inversion:** Depends on abstractions (repositories)

Design Patterns Used:

- **Facade Pattern:** Unified business logic interface
- **Repository Pattern:** Data access abstraction
- **Factory Pattern:** Entity creation
- **Strategy Pattern:** Filtering and sorting logic

6.5 Testing Strategy

Test Coverage Goals:

- **Unit Tests:** > 80% code coverage
- **Integration Tests:** All API endpoints
- **End-to-End Tests:** Critical user flows

Testing Tools:

- **Unit Tests:** pytest, unittest.mock
- **Integration Tests:** pytest with test database
- **API Tests:** pytest with requests/httpx
- **Load Tests:** Locust or Apache JMeter

6.6 Security Considerations

Authentication & Authorization:

- JWT with short expiration times (24 hours)
- Refresh token mechanism
- Role-based access control (RBAC)

Data Protection:

- Password hashing with bcrypt (cost factor 12+)
- HTTPS only in production
- SQL injection prevention (parameterized queries)
- XSS protection (input sanitization)

Rate Limiting:

- Per IP: 100 requests/minute
- Per user: 1000 requests/hour

Input Validation:

- Server-side validation (never trust client)
- Whitelist approach
- Length limits on all string inputs

6.7 Performance Optimization

Database:

- Proper indexing on frequently queried fields
- Connection pooling (10-20 connections)
- Query optimization with EXPLAIN ANALYZE
- Avoid N+1 queries with eager loading

Caching Strategy:

- Redis for session data
- Cache frequently accessed data
- Cache invalidation on updates
- TTL-based expiration

API Performance:

- Pagination for list endpoints (max 100 items)
- Gzip compression for responses
- Async operations for heavy tasks

6.8 References

Design Patterns:

- "Design Patterns: Elements of Reusable Object-Oriented Software" by Gang of Four
- "Patterns of Enterprise Application Architecture" by Martin Fowler

Architecture:

- "Clean Architecture" by Robert C. Martin
- "Domain-Driven Design" by Eric Evans

API Design:

- "REST API Design Rulebook" by Mark Masse

Python Best Practices:

- "Effective Python" by Brett Slatkin
- "Fluent Python" by Luciano Ramalho

---

7. Appendices

Appendix A: Glossary

- **API:** Application Programming Interface
- **CRUD:** Create, Read, Update, Delete
- **JWT:** JSON Web Token
- **ORM:** Object-Relational Mapping
- **REST:** Representational State Transfer
- **UUID:** Universally Unique Identifier
- **RBAC:** Role-Based Access Control
- **XSS:** Cross-Site Scripting
- **SQL:** Structured Query Language
- **HTTP:** Hypertext Transfer Protocol

Appendix B: API Endpoint Summary

Method	Endpoint	Description	Auth Required
POST	`/api/v1/users`	Create user	No
POST	`/api/v1/auth/login`	Login	No
GET	`/api/v1/users/{id}`	Get user	Yes
PUT	`/api/v1/users/{id}`	Update user	Yes (owner)

DELETE	`/api/v1/users/{id}`	Delete user	Yes (owner)
GET	`/api/v1/places`	List places	No
POST	`/api/v1/places`	Create place	Yes
GET	`/api/v1/places/{id}`	Get place	No
PUT	`/api/v1/places/{id}`	Update place	Yes (owner)
DELETE	`/api/v1/places/{id}`	Delete place	Yes (owner)
GET	`/api/v1/reviews`	List reviews	No
POST	`/api/v1/reviews`	Create review	Yes
PUT	`/api/v1/reviews/{id}`	Update review	Yes (author)
DELETE	`/api/v1/reviews/{id}`	Delete review	Yes (author)
GET	`/api/v1/amenities`	List amenities	No
POST	`/api/v1/amenities`	Create amenity	Yes (admin)

Appendix C: Database Schema

```
-- Users table
CREATE TABLE users (
   id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
   email VARCHAR(255) UNIQUE NOT NULL,
   password_hash VARCHAR(255) NOT NULL,
   first_name VARCHAR(100) NOT NULL,
   last_name VARCHAR(100) NOT NULL,
   is_admin BOOLEAN DEFAULT FALSE,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
-- Places table
CREATE TABLE places (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    owner_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    title VARCHAR(100) NOT NULL,
    description TEXT,
    price_per_night DECIMAL(10,2) NOT NULL CHECK (price_per_night > 0),
    latitude DECIMAL(10,8) CHECK (latitude BETWEEN -90 AND 90),
    longitude DECIMAL(11,8) CHECK (longitude BETWEEN -180 AND 180),
    max_guests INTEGER NOT NULL CHECK (max_guests > 0),
    number_of_rooms INTEGER DEFAULT 0,
    number_of_bathrooms INTEGER DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Reviews table
CREATE TABLE reviews (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    place_id UUID NOT NULL REFERENCES places(id) ON DELETE CASCADE,
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    rating INTEGER NOT NULL CHECK (rating BETWEEN 1 AND 5),
    text TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, place_id)
);

-- Amenities table
CREATE TABLE amenities (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name VARCHAR(50) UNIQUE NOT NULL,
    icon_url VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Place-Amenity join table
CREATE TABLE place_amenities (
    place_id UUID NOT NULL REFERENCES places(id) ON DELETE CASCADE,
    amenity_id UUID NOT NULL REFERENCES amenities(id) ON DELETE CASCADE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (place_id, amenity_id)
);
```

```
-- Indexes for performance
CREATE INDEX idx_user_email ON users(email);
CREATE INDEX idx_place_owner ON places(owner_id);
CREATE INDEX idx_place_location ON places(latitude, longitude);
CREATE INDEX idx_place_price ON places(price_per_night);
CREATE INDEX idx_review_place ON reviews(place_id);
CREATE INDEX idx_review_user ON reviews(user_id);
CREATE INDEX idx_pa_place ON place_amenities(place_id);
CREATE INDEX idx_pa_amenity ON place_amenities(amenity_id);
```

Appendix D: Environment Configuration

Development Environment (.env.dev):

```
# Database
DATABASE_URL=postgresql://hbnb_dev:dev_password@localhost:5432/hbnb_dev
DATABASE_POOL_SIZE=10

# JWT
JWT_SECRET_KEY=your-development-secret-key-change-in-production
JWT_EXPIRATION_HOURS=24

# API
API_HOST=0.0.0.0
API_PORT=5000
DEBUG=True

# Redis
REDIS_URL=redis://localhost:6379/0

# Logging
LOG_LEVEL=DEBUG
```

Production Environment (.env.prod):

```
# Database
DATABASE_URL=postgresql://hbnb_prod:strong_password@db-server:5432/hbnb_prod
DATABASE_POOL_SIZE=20

# JWT
JWT_SECRET_KEY=your-super-secret-production-key-min-32-chars
JWT_EXPIRATION_HOURS=24
```

```
# API
API_HOST=0.0.0.0
API_PORT=8000
DEBUG=False

# Redis
REDIS_URL=redis://redis-server:6379/0

# Logging
LOG_LEVEL=INFO
```

Appendix E: Sample Data for Testing

Sample Users:

```
[
  {
    "email": "john.doe@example.com",
    "password": "SecurePass123!",
    "first_name": "John",
    "last_name": "Doe"
  },
  {
    "email": "jane.smith@example.com",
    "password": "SecurePass456!",
    "first_name": "Jane",
    "last_name": "Smith"
  }
]
```

Sample Amenities:

```
[
  {"name": "WiFi", "icon_url": "https://example.com/icons/wifi.png"},
  {"name": "Swimming Pool", "icon_url": "https://example.com/icons/pool.png"},
  {"name": "Parking", "icon_url": "https://example.com/icons/parking.png"},
  {"name": "Kitchen", "icon_url": "https://example.com/icons/kitchen.png"},
  {"name": "Air Conditioning", "icon_url": "https://example.com/icons/ac.png"}
]
```

Sample Place:

```
{
  "title": "Cozy Downtown Apartment",
  "description": "Beautiful 2-bedroom apartment in the heart of Paris",
  "price_per_night": 85.00,
  "latitude": 48.8566,
  "longitude": 2.3522,
  "max_guests": 4,
  "number_of_rooms": 2,
  "number_of_bathrooms": 1,
  "owner_id": "user-uuid-123",
  "amenities": ["amenity-uuid-1", "amenity-uuid-2"]
}
```

Document Completion Checklist

✅ **High-Level Package Diagram** - Complete (Mermaid Diagram 1)
✅ **Detailed Class Diagram** - Complete (Mermaid Diagram 2)
✅ **User Registration Sequence Diagram** - Complete (Mermaid Diagram 3)
✅ **Place Registration Sequence Diagram** - Complete (Mermaid Diagram 4)
✅ **Review Submission Sequence Diagram** - Complete (Mermaid Diagram 5)
✅ **Fetching Places List Sequence Diagram** - Complete (Mermaid Diagram 6)
✅ **All Documentation Sections** - Complete
✅ **Database Schema** - Complete
✅ **API Endpoints Summary** - Complete
✅ **Design Decisions** - Complete
✅ **Implementation Roadmap** - Complete

End of Document