숙소 및 활동 예약 플랫폼

팀 명 : HPNY

지 도 교 수 : 이병천 교수님

팀 장 : 박진아

팀 원 : 노수빈

: 한유정

: 양유나

2024. 10. 24

중부대학교 정보보호학과

<목차>

1. 서론

1.1 연구 배경

[그림 1]

코로나19 팬데믹 이전부터 숙박 예약 애플리케이션의 이용률은 꾸준히 증가하고 있다.

대중적인 숙박 예약 애플리케이션을 이용하며 처음 이용하는 사용자가 원하는 기능을 직관적으로 파악하는 데 시간이 소요되는 문제점을 인식했다. 이는 사용자가 서비스를 이용하는 데 부정적인 요인으로 작용할 수 있고, 서비스 이용 시간을 감소시키는 결과를 초래할 것이다.

따라서, Comma는 사용자 친화적인 UI 제공을 통해 사용자 경험을 향상하고 숙박 예약 외의 체험활동 예약과 같은 기능을 추가하여 기존보다 다양한 기능을 사용할 수 있도록 의도하였다.

1.2 연구 목적 및 필요성

Comma의 개발 목적은 기존 숙박 예약 UI의 구조를 개선하여 사용자 경험을 향상하는 것이다. 또한 결제 시스템 간소화로 기존 결제 시스템의 복잡성을 일부 해소하고 앱 사용자의 체류 시간을 늘리는 것을 목표로 한다.

개선 사항에 대한 기대효과로 더 많은 사용자 유입과 이에 따른 사용자-제휴 업체 간에 교류 활성화가 예상된다. 구조적인 개선을 통해 기존 사용자의 이탈을 최대한 방지하고, 신규 사용자의 유입을 통해 확장된 서비스가 구성될 것이다.

2. 관련 연구

2.1 Next.JS

Next.js는 React 기반의 프레임워크다. 서버 사이드 렌더링과 정적 사이트 생성을 지원한다. 서버 사이드 렌더링은 서버에서 페이지를 렌더링하여 완전히 구성된 HTML을 클라이언트로 전송하는 방식이다. 정적 사이트 생성은 빌드 시점에 미리 페이지를 렌더링해 정적인 HTML 파일을 생성하는 방식이다. 이러한 방식은 검색 엔진 최적화와 페이지 로딩 속도 개선 등 서비스 최적화에 도움이 된다. 라우팅 방식으로는 App Router와 Pages Router 두 가지를 제공한다.

2.2 Typescript

TypeScript는 정적 타입 언어인 동시에 컴파일 언어다. 이러한 특징으로 코드의 안정성을 높일 수 있고, 컴파일 과정에서 오류를 줄일 수 있다. 또한, Typesciprt는 JavaScript의 슈퍼셋으로 JavaScript와의 호환성이 뛰어나다는 특징도 있다.

2.3 Recoil

Recoil은 페이스북에서 만든 React 상태 관리 라이브러리다. Recoil은 컴포넌트 안에서 상태를 관리할 수 있다. 또한, 비동기 처리와 개발자 도구(Dev Tools)를 지원한다. 비동기 상태 관리는 상태 변화를 자동으로 감지하여 컴포넌트를 업데이트 시킬 수 있다. 개발자 도구는 상태 변화를 추적하고 디버깅하는 데 사용된다.

2.4 Supabase

Supabase는 오픈 소스의 실시간 데이터베이스를 제공하는 플랫폼이다. PostgreSQL을 기반으로 하며 확장성과 안정성이 뛰어나다. 기능으로는 실시간 데이터 동기화, REST API, WebSocket, 인증 및 권한 관리, 스토리지 등 다양한 기능이 있다. 이 밖에도 클라우드 서비스를 제공해 서버를 호스팅하고 관리할 수 있다.

2.5 Cloudinary

Cloudinary는 이미지와 동영상을 관리, 최적화, 전송할 수 있는 플랫폼을 제공하는 클라우드 기반 서비스다. 제공하는 기능으로는 이미지 및 동영상 파일의 업로드, 저장, 관리, 자르기, URL 기반 변환, 자동 형식 변환 및 최적화, 글로벌 CDN을 통한 빠른 전송 등이 있다.

2.6 Prisma

Prisma는 Node.js 및 Typescript용 오픈소스 ORM이다. 데이터베이스와의 상호작용을 쉽고 효율적으로 할 수 있게 도와준다. SQL 쿼리문을 직접 작성하지 않아도 되며, 마이그레이션 관리로 생산성을 높일 수 있다. Prisma는 Prisma Client, Prisma Migrate, Prisma Studio라는 세 가지 도구를 제공한다. 세 도구는 각각 접근 라이브러리, 마이그레이션 도구, Prisma GUI이다.

2.7 Vercel

Vercel은 배포 및 호스팅을 쉽게 해주는 클라우드 플랫폼이다. Next.js의 개발사에서 제공하는 호스팅 서비스이며, 자동화된 배포 파이프라인을 통해 코드 변경 사항을 쉽게 적용 및 배포할 수 있다.

3. 본론

3.1 서비스 구상

Comma는 서비스 이용자에게 숙소나 활동을 예약 또는 판매할 수 있도록 한다. 사용자는 숙소 또는 활동을 검색하고 예약할 수 있으며, 판매자 자신의 숙소나 활동을 등록하고 관리할 수 있다. 로그인/회원가입 과정에서 사용자와 판매자로 나눠서 접속이 가능하며, 이 밖에도 사용자는 숙소 및 활동의 후기 작성, 찜, 공유, 지도 기능을 통한 검색이 가능하다.

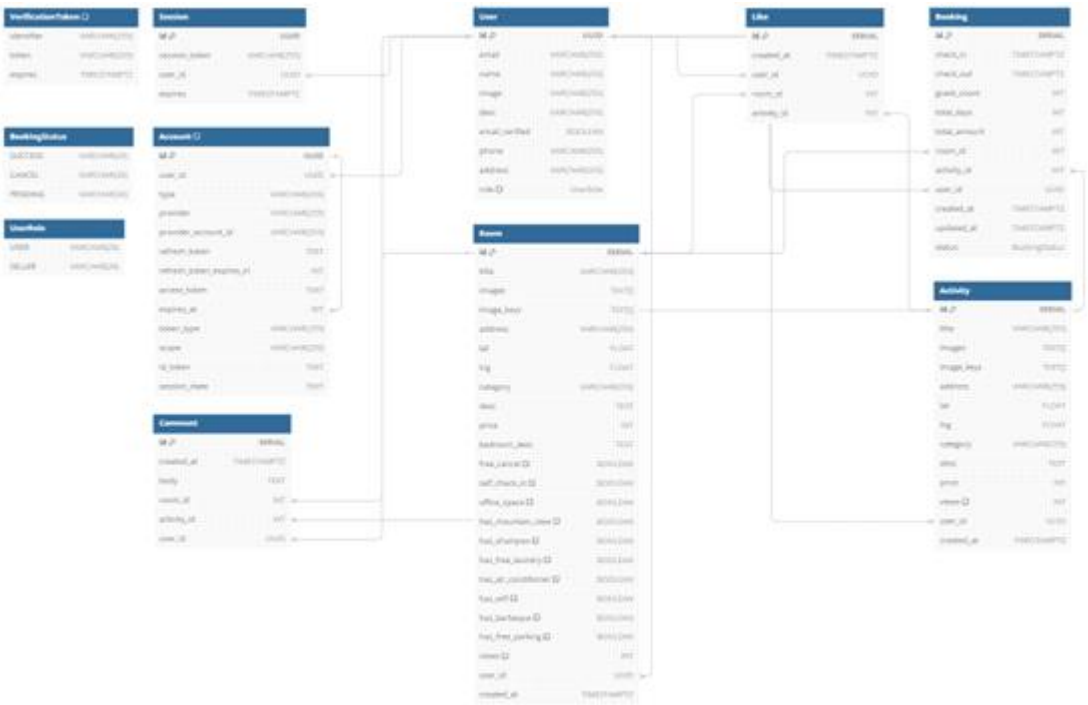
서비스는 Next.js와 Typescript, Recoil, Supabase 등 다양한 기술을 이용해 제작하는 것으로 구상했다. 클라이언트의 요청은 백엔드로 전달되어 Supabase를 통해 데이터베이스와 연결되도록 했으며, 데이터베이스는 숙소나 활동 정보, 사용자 프로필, 예약 내역 등을 저장하고 관리하도록 한다. 이러한 최신 기술들을 이용해 사용자 경험을 높이는 것을 우선으로 구상했다.

이 서비스는 사용자와 판매자 간의 안전한 거래 보장과 편리한 사용자 경험을 목표로 한다. 지속적인 피드백을 통해 서비스의 품질을 개선하고 사용자 요구에 맞춰 발전할 수 있도록 노력할 것이다.

[그림 2]는 서비스에 대한 플로우 차트이다.

[그림 2]

[그림 3]은 DB에 대한 ERD(Entity Relationship Diagram)이다.



[그림 3]

3.2 개발 환경

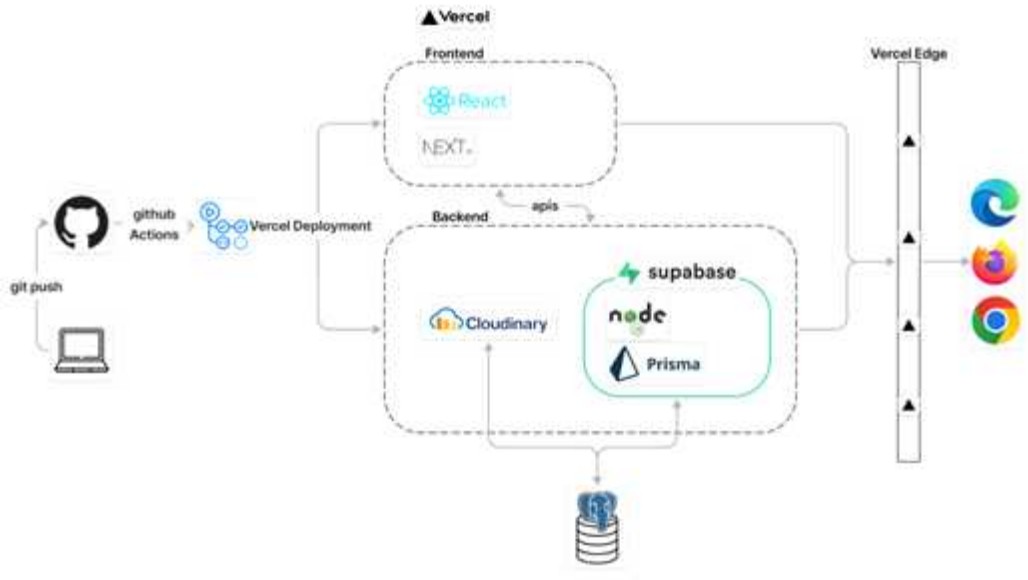
분류	내용
프로그래밍 언어	Typescript
프레임워크/런타임	Node.js, Next.js, Recoil
데이터베이스	Supabase, Cloudinary, Prisma
인프라	Vercel
도구	Git, VSCode

3. 아키텍처 구상

[그림 4]는 전체 시스템의 아키텍처이다. 사용자는 웹 브라우저를 통해 서비스를 이용하며, 프론트엔드는 Next.js로 개발되어 Vercel 플랫폼에 배포된다. 클라이언트의 요청은 Vercel의 Edge 네트워크를 통해 백엔드로 전달된다.

백엔드는 Supabase와 Node.js, Cloudinary를 사용해 구축했다. Supabase는 PostgreSQL 데이터베이스를 기반으로 사용자 데이터 저장 및 관리 기능을 제공한다. Node.js는 클라이언트의 요청을 처리하고 데이터베이스와 상호작용을 하며 API 서버 역할을 한다. Prisma는 데이터베이스 쿼리를 쉽게 관리할 수 있도록 한다. 이미지 파일 관리(업로드, 저장 등)는 Cloudinary를 사용한다. 이를 통해 성능을 최적화하고 사용자에게 이미지를 빠르게 제공한다. 코드 변경 사항은 GitHub에서 관리하며, GitHub Actions를 통해 자동 배포 프로세스가 설정되어 있어 코드 변경을 신속하게 배포할 수 있도록 한다. 사용자는 Vercel의 Edge 네트워크를 통해 웹 애플리케이션을 사용한다.

확장성과 성능을 고려해 [그림 4]와 같은 아키텍처를 설계했다. 클라이언트와 서버 간의 원활한 통신과 편리한 사용자 경험을 시키는 것을 목표로 한다.



[그림 4]

4. 서비스 소개

4.1 웹 서비스 구성

웹 서비스 주소: https://hpny-project.vercel.app/

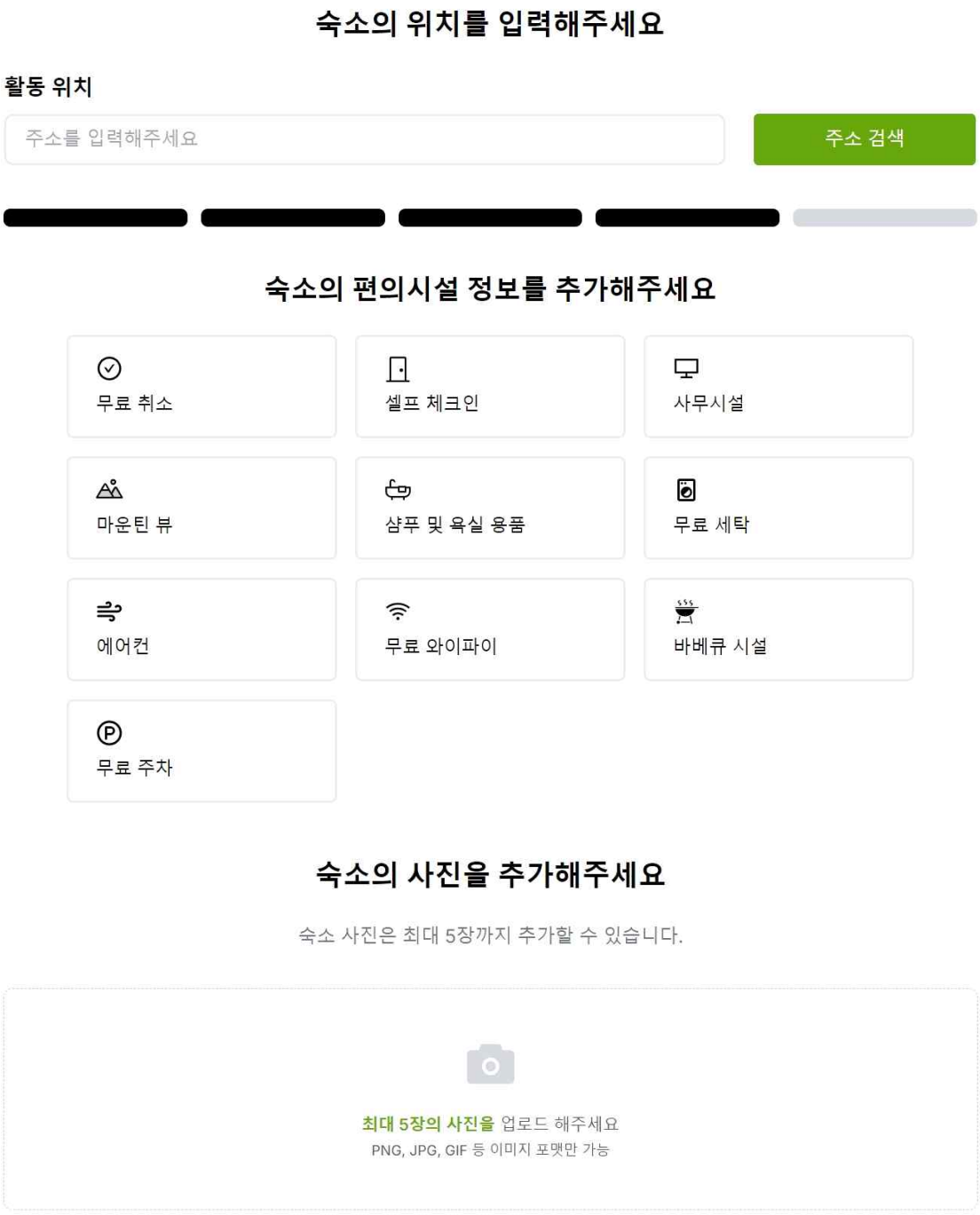
서비스 메뉴

- 로그인, 회원가입
- 숙소: 숙소를 예약하거나 등록한다.
- 활동: 활동을 예약하거나 등록한다.
- 마이페이지: 예약내역, 찜 내역을 확인한다.
- 후기: 후기를 작성할 수 있다.
- 지도: 숙소를 지도에 표시한다.

4.2 숙소/활동 등록

4.2.1 숙소/활동 등록 UI

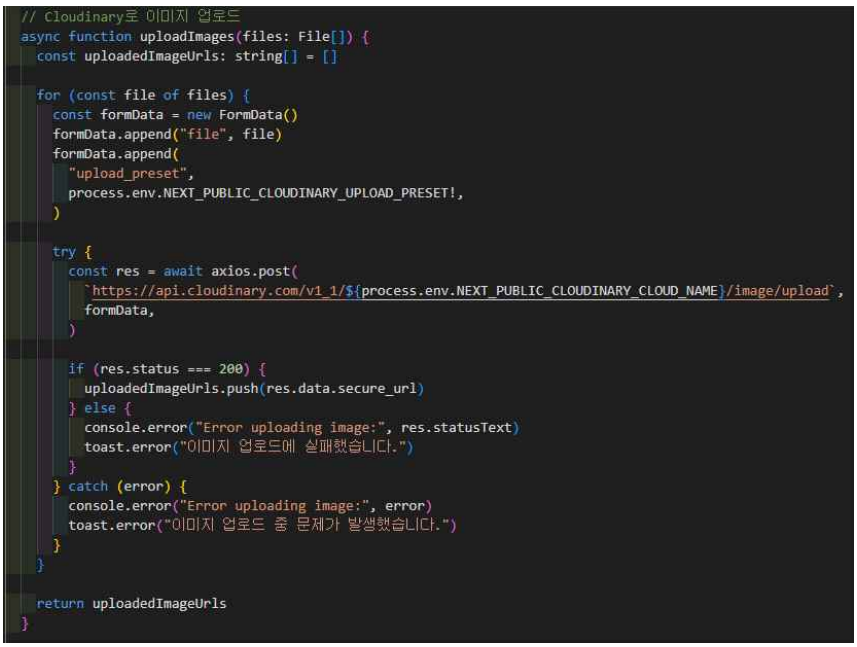
[그림 5]는 숙소 등록 과정 UI다. 활동 등록 또한 해당 UI와 동일하다.


숙소의 위치를 입력해주세요
활동 위치
주소를 입력해주세요
주소 검색
숙소의 편의시설 정보를 추가해주세요
무료 취소
셀프 체크인
사무시설
마운틴 뷰
샴푸 및 욕실 용품
무료 세탁
에어컨
무료 와이파이
바베큐 시설
무료 주차
숙소의 사진을 추가해주세요
숙소 사진은 최대 5장까지 추가할 수 있습니다.
최대 5장의 사진을 업로드 해주세요
PNG, JPG, GIF 등 이미지 포맷만 가능

[그림 5]

4.2.2 숙소/활동 등록 소스코드

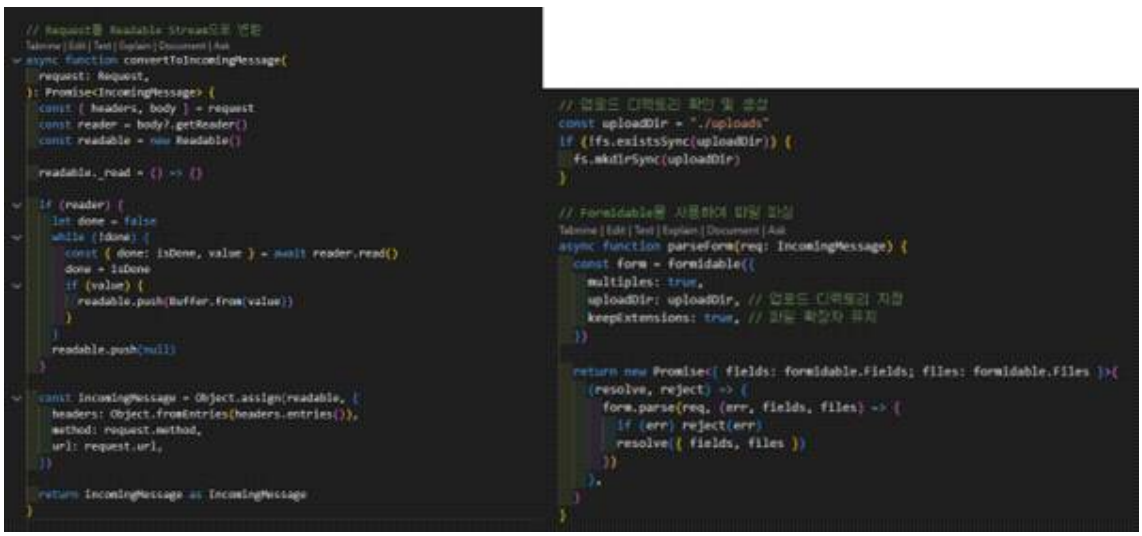
[그림 6]은 숙소/활동 등록 중 이미지 처리에 대한 함수이다.



```
// Cloudinary로 이미지 업로드
async function uploadImages(files: File[]) {
  const uploadedImageUrls: string[] = []

  for (const file of files) {
    const formData = new FormData()
    formData.append("file", file)
    formData.append(
      "upload_preset",
      process.env.NEXT_PUBLIC_CLOUDINARY_UPLOAD_PRESET!,
    )

    try {
      const res = await axios.post(
        `https://api.cloudinary.com/v1_1/${process.env.NEXT_PUBLIC_CLOUDINARY_CLOUD_NAME}/image/upload`,
        formData,
      )

      if (res.status === 200) {
        uploadedImageUrls.push(res.data.secure_url)
      } else {
        console.error("Error uploading image:", res.statusText)
        toast.error("이미지 업로드에 실패했습니다.")
      }
    } catch (error) {
      console.error("Error uploading image:", error)
      toast.error("이미지 업로드 중 문제가 발생했습니다.")
    }
  }

  return uploadedImageUrls
}
```

[그림 6]

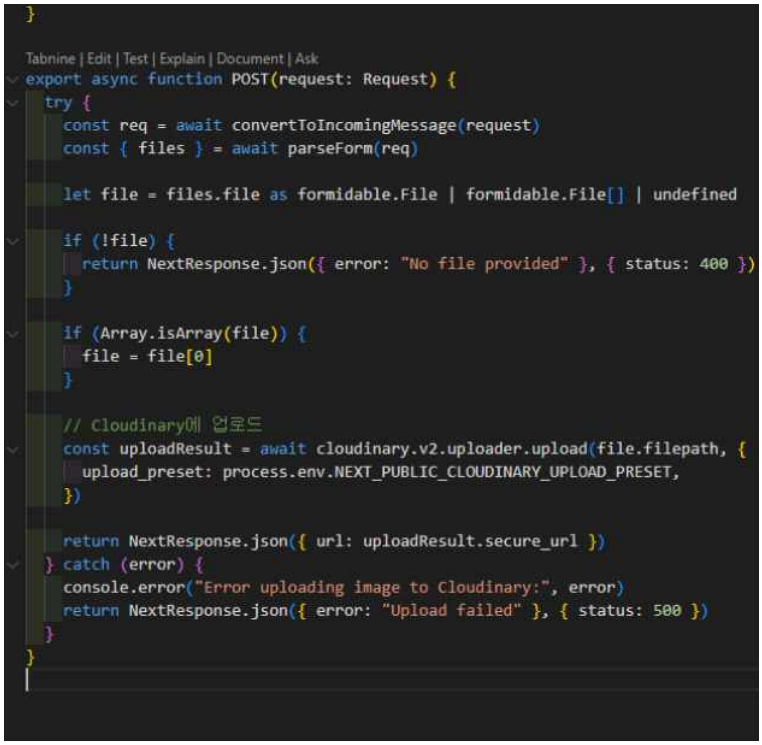
[그림 7], [그림 8]는 이미지 파일 처리에 관한 로직이다. 우선 클라이언트가 요청을 보내면, 서버는 convertToIncomingMessage 함수를 호출해 Readable Stream으로 변환한다. 이후 parseForm 함수를 통해 업로드된 파일을 파싱하고, 디렉터리를 생성한다. 파싱된 파일은 Cloudinary에 업로드되고, 성공 시 URL을 포함한 JSON 응답을 반환한다.



```
// Request를 Readable Stream으로 변환
async function convertToIncomingMessage(
  request: Request,
): Promise<IncomingMessage> {
  const { headers, body } = request
  const reader = body?.getReader()
  const readable = new Readable()

  readable._read = () => {}

  if (reader) {
    let done = false
    while (!done) {
      const { done: isDone, value } = await reader.read()
      done = isDone
      if (value) {
        readable.push(Buffer.from(value))
      }
    }
    readable.push(null)
  }

  const IncomingMessage = Object.assign(readable, {
    headers: Object.fromEntries(headers.entries()),
    method: request.method,
    url: request.url,
  })

  return IncomingMessage as IncomingMessage
}
```

```
// 업로드 디렉토리 확인 및 생성
const uploadDir = "./uploads"
if (!fs.existsSync(uploadDir)) {
  fs.mkdirSync(uploadDir)
}

// Formidable을 사용하여 파일 파싱
async function parseForm(req: IncomingMessage) {
  const form = formidable({
    multiples: true,
    uploadDir: uploadDir, // 업로드 디렉토리 지정
    keepExtensions: true, // 파일 확장자 유지
  })

  return new Promise<{ fields: formidable.Fields; files: formidable.Files }>(
    (resolve, reject) => {
      form.parse(req, (err, fields, files) => {
        if (err) reject(err)
        resolve({ fields, files })
      })
    },
  )
}
```

[그림 7]

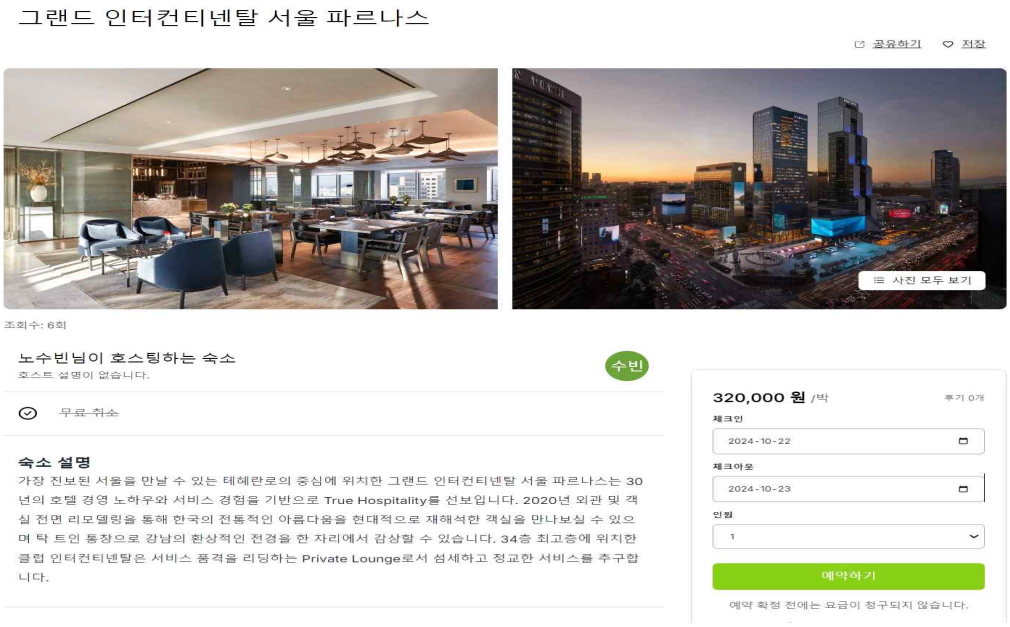


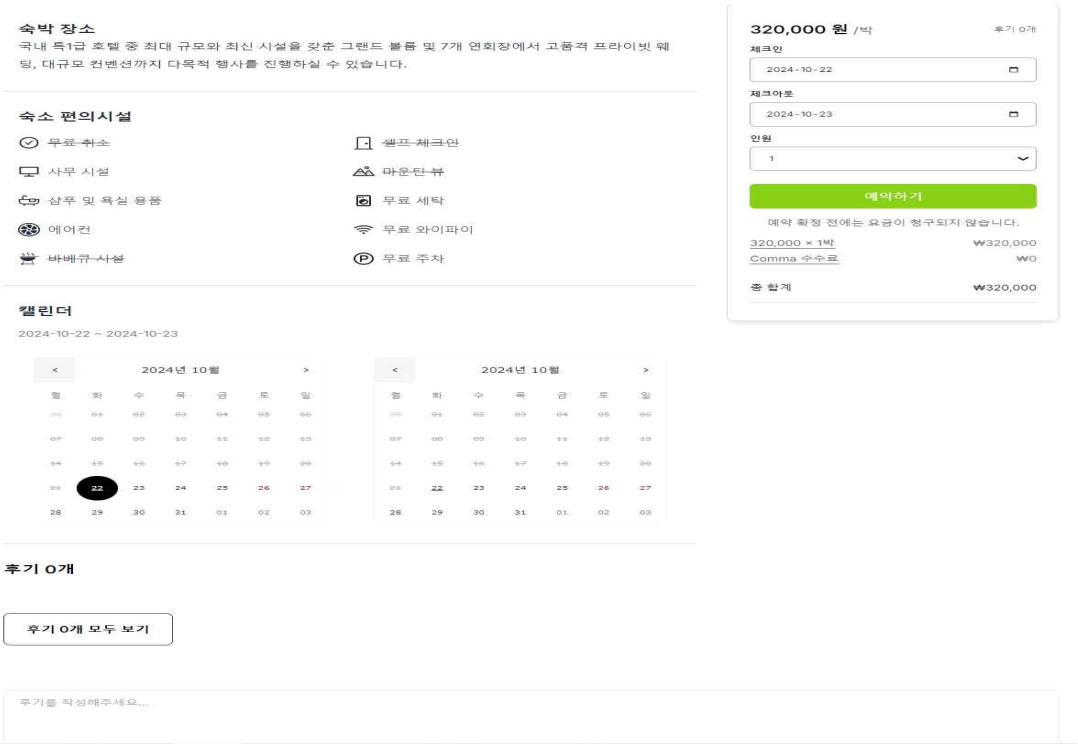
```
}

Tabnine | Edit | Test | Explain | Document | Ask
export async function POST(request: Request) {
  try {
    const req = await convertToIncomingMessage(request)
    const { files } = await parseForm(req)

    let file = files.file as formidable.File | formidable.File[] | undefined

    if (!file) {
      return NextResponse.json({ error: "No file provided" }, { status: 400 })
    }

    if (Array.isArray(file)) {
      file = file[0]
    }

    // Cloudinary에 업로드
    const uploadResult = await cloudinary.v2.uploader.upload(file.filepath, {
      upload_preset: process.env.NEXT_PUBLIC_CLOUDINARY_UPLOAD_PRESET,
    })

    return NextResponse.json({ url: uploadResult.secure_url })
  } catch (error) {
    console.error("Error uploading image to Cloudinary:", error)
    return NextResponse.json({ error: "Upload failed" }, { status: 500 })
  }
}
```

[그림 8]

4.3 숙소/활동 예약 및 상세

[그림 9]는 숙소의 상세 페이지 UI이다. 활동 상세 페이지 또한 해당 UI와 동일하다.

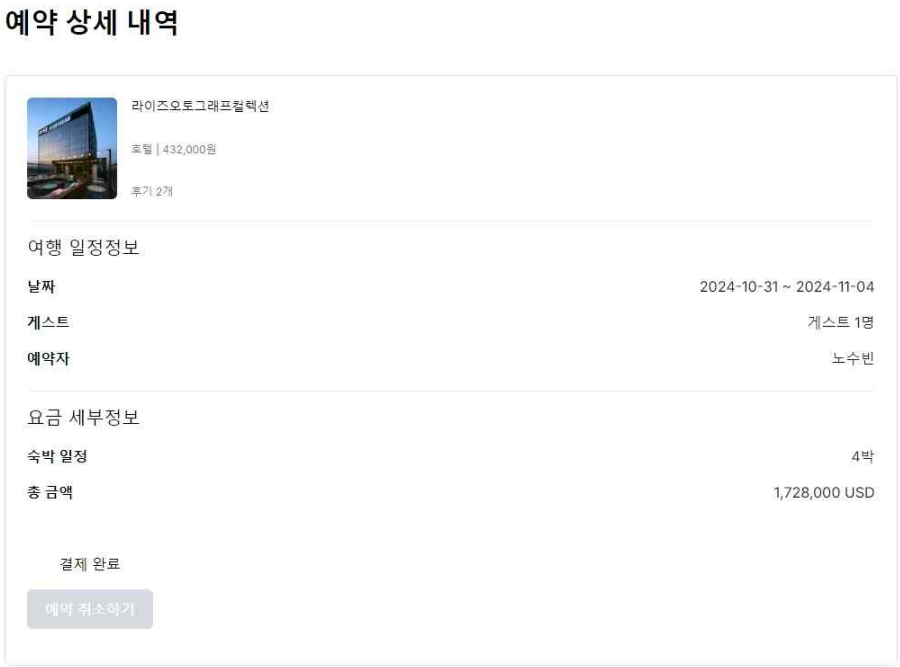




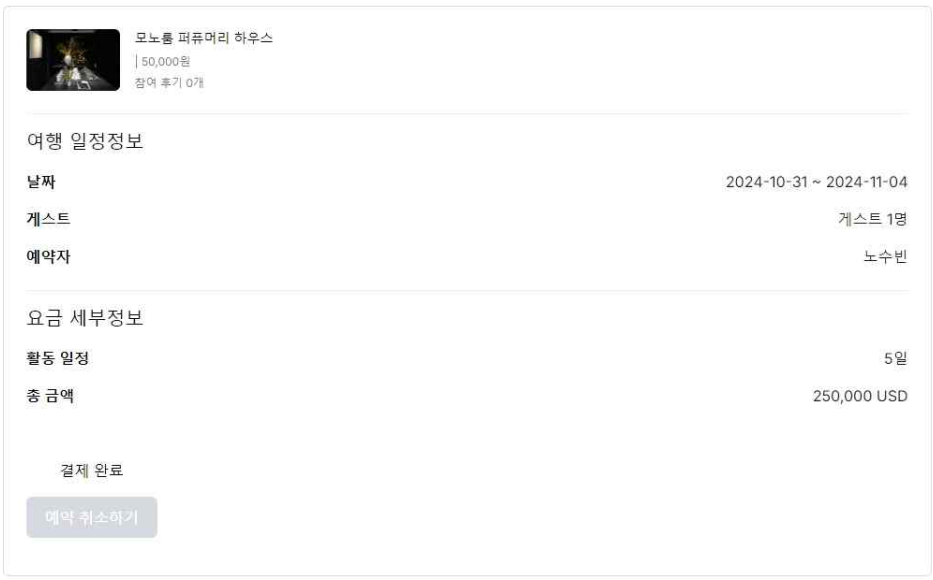
[그림 9]

4.4 숙소/활동 예약 확인

[그림 10]은 각각 숙소와 활동의 예약 확인 페이지 UI다. 결제는 PayPal의 가상 결제를 활용했다.



예약 상세 내역



[그림 10]

4.5 계정

4.5.1 사용자 계정

[그림 11]은 사용자 계정 상세 페이지 UI이다. [그림 12]는 메인 페이지 로고가 Comma로 나타내는 이미지이다.

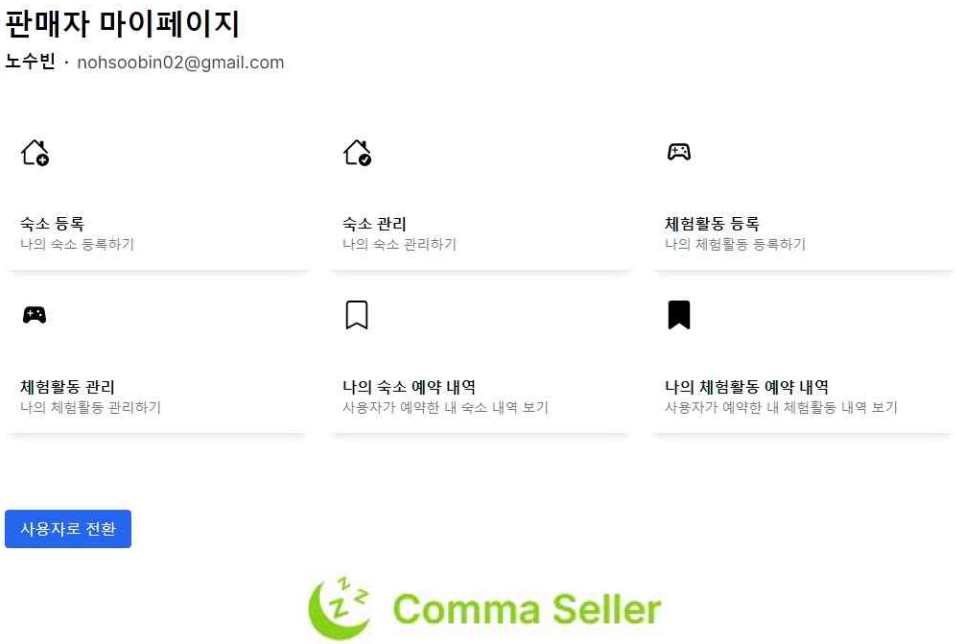
- 개인정보: 계정 관련 개인정보들이다.
- 찜 한 숙소 및 활동: 찜한 숙소나 활동을 확인한다.
- 나의 댓글: 숙소 및 활동에 단 나의 댓글을 확인한다.
- 나의 예약: 예약한 숙소 및 활동을 확인한다.

4.5.2 판매자 계정

[그림 13]은 판매자 계정 상세 페이지 UI이다. [그림 14]는 메인 페이지 로고가 Comma Seller로 나타내는 이미지이다.

- 숙소 등록: 숙소를 등록한다.
- 숙소 관리: 등록한 숙소를 관리한다.
- 활동 등록: 활동을 등록한다.
- 활동 관리: 활동을 관리한다.
- 나의 숙소 예약 내역: 예약된 숙소를 확인한다.
- 나의 활동 예약 내역: 예약된 활동을 확인한다.

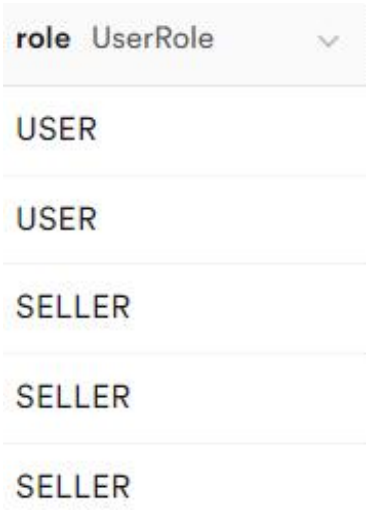
[그림 11], [그림 12]



[그림 13], [그림 14]

4.5.3 계정 소스코드

User의 경우 [그림 15]와 같이 사용자와 판매자를 구분한다.



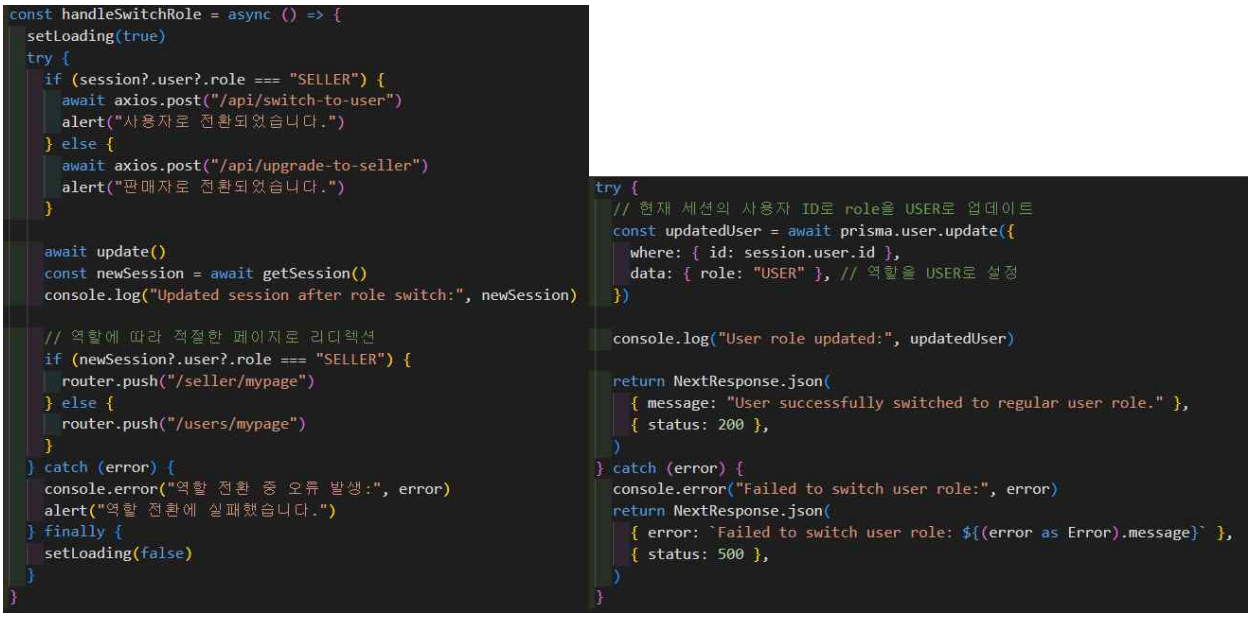
[그림 15]

[그림 16]은 사용자의 역할을 조회하는 API 핸들러이다. 우선 getServerSession 메서드를 사용해 현재 사용자의 세션을 가져온다. 그 후 요청 URL에서 userId를 추출해 prisma.user.findUnique 메서드를 사용하고 userId에 해당하는 사용자를 조회한다. 이때 역할 정보만 가져온다.

```
// 사용자 역할 조회 API 핸들러
Tabnine | Edit | Test | Explain | Document | Ask
export async function GET(req: Request) {
  const session = await getServerSession(authOptions)

  // 세션이 없으면 403 상태 반환
  if (!session) {
    return NextResponse.json({ error: "Unauthorized access." }, { status: 403 })
  }

  // 요청 URL에서 userId 가져오기
  const { searchParams } = new URL(req.url)
  const userId = searchParams.get("userId")

  // userId가 없으면 400 상태 반환
  if (!userId) {
    return NextResponse.json({ error: "User ID is required." }, { status: 400 })
  }

  try {
    // 사용자 조회
    const user = await prisma.user.findUnique({
      where: { id: userId },
      select: { role: true }, // role만 선택적으로 조회
    })

    if (user) {
      // 사용자 역할 반환
      return NextResponse.json({ initialRole: user.role }, { status: 200 })
    } else {
      // 사용자가 존재하지 않으면 404 상태 반환
      return NextResponse.json({ error: "User not found." }, { status: 404 })
    }
  } catch (error) {
    console.error("Error fetching user role:", error)
    // 서버 오류 발생 시 500 상태 반환
    return NextResponse.json(
      { error: "Internal server error" },
      { status: 500 },
    )
  }
}
```

[그림 16]

[그림 17]은 각각 유저의 역할(UserRole)을 사용자로 전환하기 위한 호출 함수와 로직이다. prisma.user.update 메서드를 호출해 데이터베이스에서 해당 사용자의 역할을 USER로 변경한다. 업데이트가 성공하면, 업데이트된 사용자 정보를 updatedUser변수에 저장한다.

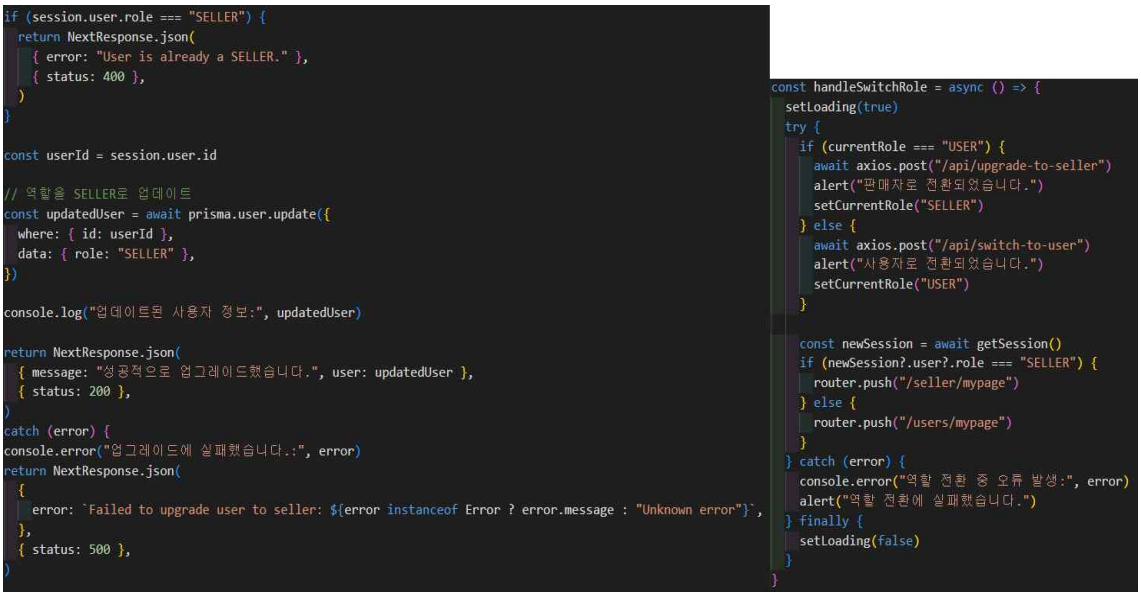


```
const handleSwitchRole = async () => {
  setLoading(true)
  try {
    if (session?.user?.role === "SELLER") {
      await axios.post("/api/switch-to-user")
      alert("사용자로 전환되었습니다.")
    } else {
      await axios.post("/api/upgrade-to-seller")
      alert("판매자로 전환되었습니다.")
    }

    await update()
    const newSession = await getSession()
    console.log("Updated session after role switch:", newSession)

    // 역할에 따라 적절한 페이지로 리디렉션
    if (newSession?.user?.role === "SELLER") {
      router.push("/seller/mypage")
    } else {
      router.push("/users/mypage")
    }
  } catch (error) {
    console.error("역할 전환 중 오류 발생:", error)
    alert("역할 전환에 실패했습니다.")
  } finally {
    setLoading(false)
  }
}
```

```
try {
  // 현재 세션의 사용자 ID로 role을 USER로 업데이트
  const updatedUser = await prisma.user.update({
    where: { id: session.user.id },
    data: { role: "USER" }, // 역할을 USER로 설정
  })

  console.log("User role updated:", updatedUser)

  return NextResponse.json(
    { message: "User successfully switched to regular user role." },
    { status: 200 },
  )
} catch (error) {
  console.error("Failed to switch user role:", error)
  return NextResponse.json(
    { error: `Failed to switch user role: ${(error as Error).message}` },
    { status: 500 },
  )
}
```

[그림 17]

[그림 18]은 각각 유저의 역할을 판매자로 전환하기 위한 로직과 호출 함수이다. 사용자 전환과 동일하게 prisma.user.update 메서드를 통해 해당 사용자의 역할을 SELLER로 변경한다. 업데이트가 되면 사용자 정보를 updatedUser변수에 저장한다.



```
if (session.user.role === "SELLER") {
  return NextResponse.json(
    { error: "User is already a SELLER." },
    { status: 400 },
  )
}

const userId = session.user.id

// 역할을 SELLER로 업데이트
const updatedUser = await prisma.user.update({
  where: { id: userId },
  data: { role: "SELLER" },
})

console.log("업데이트된 사용자 정보:", updatedUser)

return NextResponse.json(
  { message: "성공적으로 업그레이드했습니다.", user: updatedUser },
  { status: 200 },
)
catch (error) {
console.error("업그레이드에 실패했습니다.:", error)
return NextResponse.json(
  {
    error: `Failed to upgrade user to seller: ${error instanceof Error ? error.message : "Unknown error"}`,
  },
  { status: 500 },
)
```

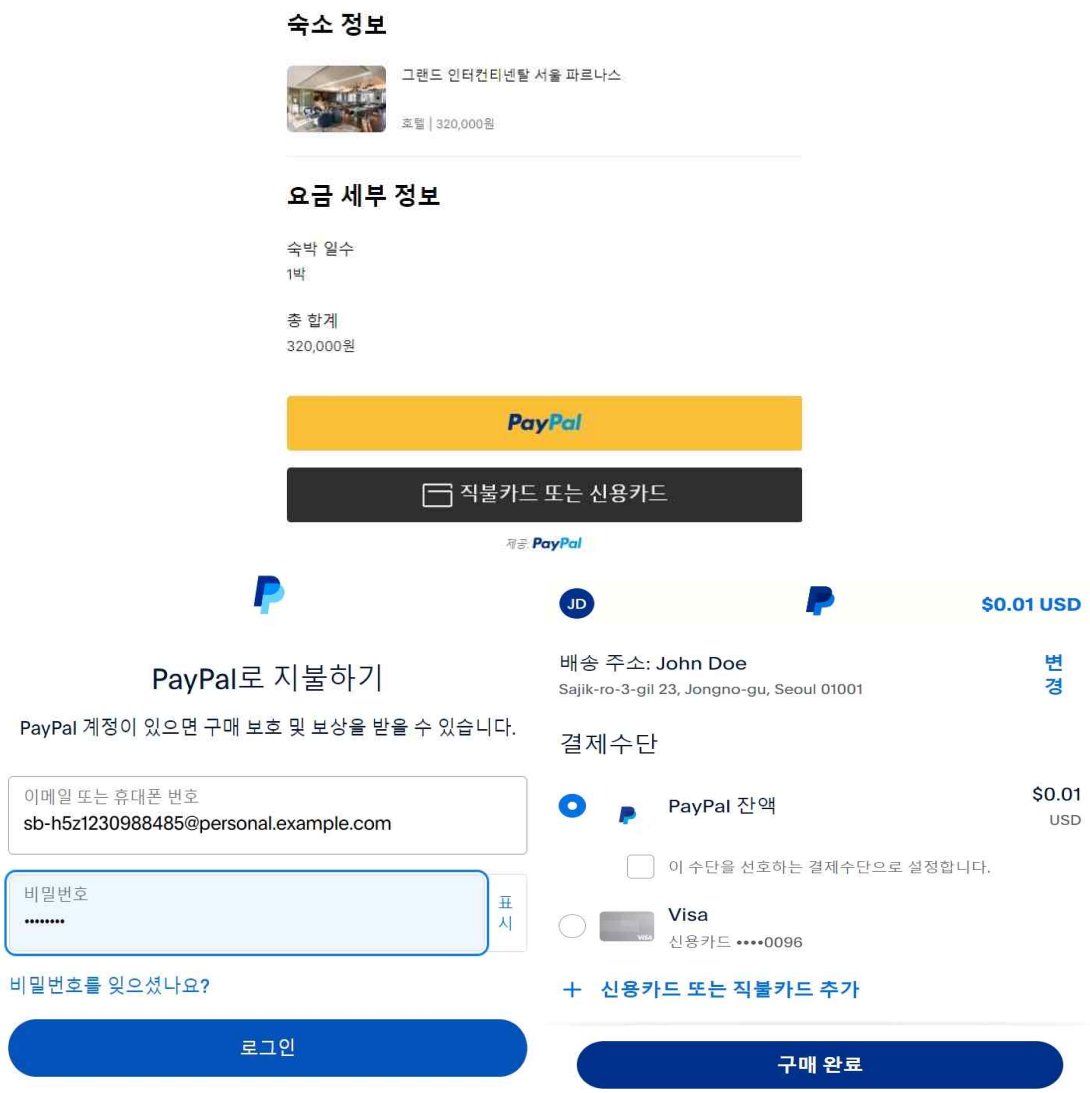
```
const handleSwitchRole = async () => {
  setLoading(true)
  try {
    if (currentRole === "USER") {
      await axios.post("/api/upgrade-to-seller")
      alert("판매자로 전환되었습니다.")
      setCurrentRole("SELLER")
    } else {
      await axios.post("/api/switch-to-user")
      alert("사용자로 전환되었습니다.")
      setCurrentRole("USER")
    }

    const newSession = await getSession()
    if (newSession?.user?.role === "SELLER") {
      router.push("/seller/mypage")
    } else {
      router.push("/users/mypage")
    }
  } catch (error) {
    console.error("역할 전환 중 오류 발생:", error)
    alert("역할 전환에 실패했습니다.")
  } finally {
    setLoading(false)
  }
}
```

[그림 18]

4.6 가상 결제

4.6.1 가상 결제

[그림 19]는 가상 결제 과정 UI이다. 가상 결제는 실제 금전 거래 없이 결제 시스템을 테스트하거나 사용자 경험을 검증하기 위해 사용하는 방법이다. PayPal은 안전한 거래를 보장하며, 가상 결제를 통해 사용자는 실제 결제 과정에서의 불편함 없이 시스템을 경험할 수 있다. 본인의 PayPal 계정을 통해 들어가서 가상 결제를 진행한다.

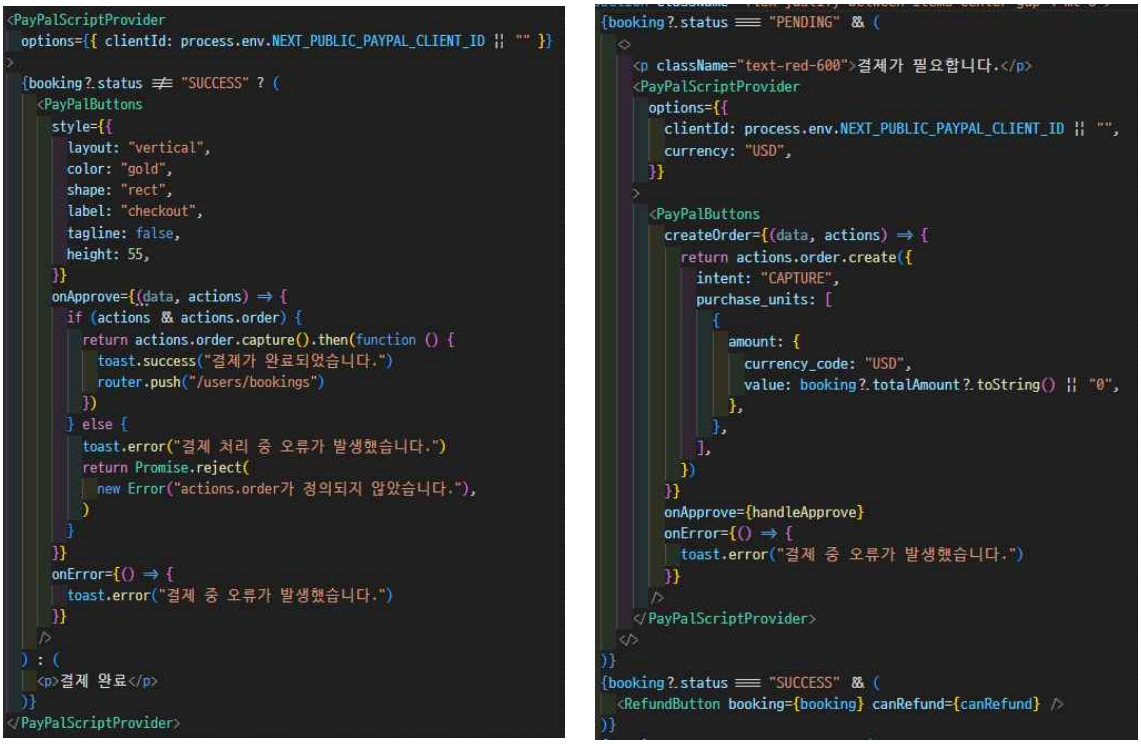


[그림 19]

4.6.2 결제 소스코드

[그림 20]은 각각 PayPal 버튼 스크립트이다. PayPalScriptProvider는 PayPal SDK를 로드하고 사용할 수 있도록 설정한다. clientId와 currency를 통해 결제 설정을 정의한다.

PayPalButtons는 결제 버튼을 생성한다. createOrder는 사용자가 결제 버튼을 클릭했을 때 호출된다. 결제 주문을 생성하며, intent는 'CAPTURE'로 설정해 결제를 실행한다.



```
<PayPalScriptProvider
  options={{ clientId: process.env.NEXT_PUBLIC_PAYPAL_CLIENT_ID ?? "" }}
>
  {booking?.status === "SUCCESS" ? (
    <PayPalButtons
      style={{
        layout: "vertical",
        color: "gold",
        shape: "rect",
        label: "checkout",
        tagline: false,
        height: 55,
      }}
      onApprove={(data, actions) => {
        if (actions && actions.order) {
          return actions.order.capture().then(function () {
            toast.success("결제가 완료되었습니다.")
            router.push("/users/bookings")
          })
        } else {
          toast.error("결제 처리 중 오류가 발생했습니다.")
          return Promise.reject(
            new Error("actions.order가 정의되지 않았습니다."),
          )
        }
      }}
      onError={() => {
        toast.error("결제 중 오류가 발생했습니다.")
      }}
    />
  ) : (
    <p>결제 완료</p>
  )}
</PayPalScriptProvider>
```

```
{booking?.status === "PENDING" && (
  <>
    <p className="text-red-600">결제가 필요합니다.</p>
    <PayPalScriptProvider
      options={{
        clientId: process.env.NEXT_PUBLIC_PAYPAL_CLIENT_ID ?? "",
        currency: "USD",
      }}
    >
      <PayPalButtons
        createOrder={(data, actions) => {
          return actions.order.create({
            intent: "CAPTURE",
            purchase_units: [
              {
                amount: {
                  currency_code: "USD",
                  value: booking?.totalAmount?.toString() ?? "0",
                },
              },
            ],
          })
        }}
        onApprove={handleApprove}
        onError={() => {
          toast.error("결제 중 오류가 발생했습니다.")
        }}
      />
    </PayPalScriptProvider>
  </>
)}
{booking?.status === "SUCCESS" && (
  <RefundButton booking={booking} canRefund={canRefund} />
)}
```

[그림 20]

5. 결론

5.1 결론 및 기대효과

Comma는 사용자 친화적인 숙박 예약 플랫폼을 개발했다. 사용자들이 편의성을 요구함에 따라 Comma는 다양한 검색 기준을 지원해 원하는 숙소를 찾도록 도와준다. 또한, 확장할 수 있는 구조로 설계되었기에 다양한 기능을 추가해 서비스의 발전을 기대할 수 있다.

이러한 점은 이용자에게 Comma를 사용할 이유를 만들게 하고, 이용자가 늘어남에 따라 서비스의 발전을 기대할 수 있다.

5.2 추후 보완 사항

Comma는 편리한 사용자 경험을 위해 여러 기능을 추가할 계획이다. 우선, 가격대, 평점, 접근성, 편의시설 등을 기준으로 검색할 수 있는 고급 검색(필터링) 기능을 추가할 예정이다. 고급 검색 기능은 빠르고 유연한 검색 기능을 제공함으로써 더욱 편리한 사용자 경험을 제공할 수 있을 것으로 예상된다.

그다음 학습 알고리즘을 활용하여 이용자의 활동이나 찜, 예약 기록을 기반으로 하는 추천 서비스를 제작할 계획이다. 이러한 학습 알고리즘은 편리한 사용자 경험을 제공할 수 있을 것이다.

마지막으로, 모바일에서 작동하도록 제작할 계획이다. Progressive Web App(PWA)로 개발해 사용자에게 앱과 같은 경험(알림 기능 등)을 제공해 이용자가 다른 디바이스에서도 서비스를 이용할 수 있게 구상 중이다.

6. 별첨

6.1 팀원 소개

박진아(팀장)

역할: 프론트엔드

개인 Github 주소: https://github.com/Parkpasss

노수빈(팀원)

역할: 백엔드

개인 Github 주소: https://github.com/nohsoobin

양유나(팀원)

역할: 프론트엔드

개인 Github 주소: https://github.com/una3325

한유정(팀원)

역할: 백엔드

개인 Github 주소: https://github.com/HanYooJa

6.2 팀 프로젝트 GitHub 주소

팀 프로젝트 GitHub 주소: https://github.com/Parkpasss/HPNY

6.3 시연 영상 주소

https://youtu.be/ilfjiqfhKUI?si=JO7G2vP2qBAporBj

6.4 발표 자료(별첨)