Guide: Integrating the Modular Appointment API

1. How it Works: A High-Level View

This API is a **headless service**. This means it's a self-contained "brain" that handles all the logic and data for appointment scheduling, but it has no user interface (UI) of its own. This is a powerful design because it allows you to build any kind of frontend—for a website, a mobile app, or an in-store kiosk—that can talk to this single, consistent backend.

The core workflow is a simple, four-step process:

1. **Admin Sets Availability:** The business owner or professional uses a secure interface to tell the API when they are available. The API then creates all the 30-minute appointment slots for those times.
2. **Customer Views Slots:** A customer visiting your website asks the API for all available slots. Your website displays these slots to the customer.
3. **Customer Reserves a Slot:** The customer clicks on a slot. Your website tells the API to place a temporary 5-minute hold on it. This prevents anyone else from booking it while the customer completes the final step.
4. **Customer Confirms Booking:** After the customer successfully makes a payment or completes a form, your website tells the API to permanently confirm the booking.

2. Integrating the API as a Modular Component

Because the API is headless, integrating it simply means making standard web requests from your website's code. This modular approach works for any platform (Shopify, WordPress, a custom React/Vue site, etc.).

Prerequisite: Your Node.js API must be deployed to a public server (e.g., using a service like Render, Heroku, or AWS) so it has a public URL (e.g., https://my-awesome-api.onrender.com).

Here is a step-by-step guide on how a typical website would integrate this API using client-side JavaScript.

Step 1: Fetch and Display Available Slots

On your booking page, use JavaScript to call the GET /slots endpoint.

```
// This function runs when your booking page loads
async function displayAppointmentSlots() {
```

```
    const calendarContainer = document.getElementById('calendar');
    try {
        // Call your deployed API's endpoint
        const response = await fetch('https://<your-api-url>/api/slots');
        const availableSlots = await response.json();

        // Clear any old slots from the display
        calendarContainer.innerHTML = '<h3>Select a Time:</h3>';

        // Loop through the results and create a button for each one
        availableSlots.forEach(slot => {
            const slotButton = document.createElement('button');
            slotButton.innerText = `${slot.date} at ${slot.time}`;
            // Store the unique slot ID on the button itself
            slotButton.dataset.slotId = slot.id;
            slotButton.onclick = () => handleSlotSelection(slot.id);
            calendarContainer.appendChild(slotButton);
        });

    } catch (error) {
        calendarContainer.innerHTML = '<p>Sorry, could not load available times.</p>';
        console.error('Failed to fetch slots:', error);
    }
}

// Run the function
displayAppointmentSlots();
```

Step 2: Handle Slot Reservation and Payment

When a user clicks a slot button, you need to reserve it and then guide them to payment.

```
async function handleSlotSelection(slotId) {
    try {
        // Step 2.1: Call the '/book' endpoint to place a 5-minute hold
        const response = await fetch(`https://<your-api-url>/api/slots/${slotId}/book`, {
            method: 'POST'
        });
```

```
        if (!response.ok) {
            // This happens if someone books the slot just before you did
            alert('Sorry, this slot just became unavailable. Please select another.');
            displayAppointmentSlots(); // Refresh the list
            return;
        }

        const bookingResult = await response.json();
        console.log(bookingResult.message); // "Slot reserved for 5 minutes..."

        // Step 2.2: Redirect to a payment page (e.g., Stripe, PayPal, or Shopify Checkout)
        // You pass the slotId to the payment page so you can confirm it later.
        window.location.href = `/checkout?paymentForSlot=${slotId}`;

    } catch (error) {
        alert('An error occurred. Please try again.');
        console.error('Failed to reserve slot:', error);
    }
}
```

Step 3: Confirm the Booking After Payment

This is the most critical step. After the payment provider (e.g., Stripe) confirms the payment was successful, your website's server (or a secure "thank you" page) must make the final call to the API.

This call should ideally be made from your server-side code to protect the confirmation process.

```
// This code would run on your server after receiving a payment confirmation webhook
// or on a secure page after the user returns from the payment gateway.

async function confirmAppointment(slotId) {
    try {
        const response = await fetch(`https://<your-api-url>/api/slots/${slotId}/confirm`, {
            method: 'POST'
        });
```

```
        const confirmationResult = await response.json();

        if (response.ok) {
            console.log('Successfully confirmed booking!');
            // Now you can show the user a final "Booking Confirmed!" message.
            // You might also trigger a confirmation email here.
        } else {
            // Handle rare case where confirmation fails (e.g., lock expired)
            console.error('Failed to confirm booking:', confirmationResult.message);
            // You may need to refund the customer in this edge case.
        }
    } catch (error) {
        console.error('Server error during confirmation:', error);
    }
}

// Example usage on a thank you page:
// const urlParams = new URLSearchParams(window.location.search);
// const confirmedSlotId = urlParams.get('confirmedSlot');
// confirmAppointment(confirmedSlotId);
```

By following this three-step pattern, you can integrate a robust appointment booking system into any website, keeping your business logic clean, separate, and reusable.