Alan AI Cheatsheet

Your quick reference for building multimodal conversational experiences with Alan AI. Covers script concepts, client API, handlers, and best practices.

Alan AI Scripting Basics

Intents & Commands

Basic Intent

```
intent('hello world', p => { ... });
```

Matches simple phrases like 'hello world'.

Intent with Wildcards

```
intent('I want *', p => { ... });
```

Matches phrases like 'I want coffee', 'I want pizza', etc.

Intent with Entities

```
intent('I want $(item P:item~)', p => { ... });
```

Matches 'I want coffee' and captures 'coffee' as an entity 'item'.

Multiple Phrases

```
intent(['What is $(item P:item~)', 'Tell me about $(item P:item~)'], p => { ... });
```

Matches either phrase.

Contextual Intent

```
intent('yes', { in: 'confirm_order' }, p => { ... });
```

Only matches 'yes' when in the 'confirm_order' context.

Command (No NLU)

```
intent({ command: 'start_game' }, p => { ... });
```

Triggered directly from the client via `alanBtn().callClient('start_game');`

Capturing Entity Value

Inside the intent handler:

```
p.item.value
```

Accesses the recognized value of the 'item' entity.

Responding with Text

```
p.play('Okay, I can do that.');
```

Alan says the text.

Responding with Sound/SSML

```
p.play('<audio src="sound.mp3"/>');
```

Plays a sound file. Alan supports SSML.

Chaining Plays

```
p.play('First part.'); p.play('Second part.');
```

Plays sequentially.

Playing from List

```
p.play(['OK.', 'Got it.']);
```

Randomly picks one response.

Setting Visual State

```
p.visual( { screen: 'items', data: itemsList } );
```

Updates the visual state on the client application.

Calling Client Function

```
p.callClient('updateCart', { item: p.item.value });
```

Triggers a handler (`onCommand`) on the client side.

Adding Follow-up Question

```
p.play('What size do you need?'); p.then(p => { ... });
```

Sets up a follow-up intent block.

Ending Conversation

```
p.play('Okay, I'm done.'); p.resolve();
```

Alan finishes speaking and resolves the current intent processing.

Handling No Match

Use the `fallback` intent at the end of your script to catch unrecognized phrases.

Best Practice: Be Specific

Define specific intents before using broad ones or fallback to avoid misinterpretations.

Best Practice: User Testing

Test with real users speaking naturally to refine your intents.

Advanced Script Concepts

Follow-ups and Contexts

Basic Follow-up

```
intent('I want a $(item P:item~)', p => {
    p.play('Okay, a ' + p.item.value + '. What size?');
    p.then(p => {
        intent('$(size P:size~)', p => {
            p.play('Got it. Size ' + p.size.value);
        });
    });
});
```

Sets up a temporary context (`p.then`) for the next user turn.

Named Contexts

```
intent('start order', p => {
    p.play('What would you like to order?');
    p.setContext('order_context');
});

intent('I want $(item P:item~)', { in: 'order_context' }, p => {
    p.play('Adding ' + p.item.value + ' to your order.');
    // Stays in 'order_context'
});
```

Use `p.setContext('context_name')` to transition to a new context, or `p.setContext('')` to go back to the global context.

Entering Context on Match

```
intent('start search', p => {
    p.play('Okay, what are you looking for?');
}, { context: 'search_context' }); // Enter context immediately after this intent matches

intent('$(query P:query~)', { in: 'search_context' }, p => {
    p.play('Searching for ' + p.query.value);
    p.setContext(''); // Exit context after processing
});
```

The `context` option in the intent definition sets the context after the intent matches and before its handler runs.

Context Lifecycle

- `p.then()` creates a temporary context for *only the very next turn*. If the user says something that doesn't match the intent within `p.then`, they exit the temporary context.
- `p.setContext('name')` creates a persistent context that remains active until explicitly changed or cleared with `p.setContext('')`.
- Intents with the `in: 'context_name'` option only activate when that specific context is active.
- Global intents (no `in` option) are always active, regardless of the current context.

Tips for Contexts

- Use contexts to manage conversation flow and disambiguate user input.
- Group related intents within the same context.
- Design contexts to guide the user through specific flows (e.g., checkout, account setup).
- Don't create too many contexts; keep it manageable.

Avoiding Context Loops

Ensure there are paths to exit contexts, either by matching a specific intent within the context that calls `p.setContext('')` or by having global intents that can interrupt and take the user elsewhere.

Accessing Context Name

`p.context` returns the name of the current active context.

Debug Tool: Context View

Use the Alan AI Studio Debugger's 'Contexts' tab to see which contexts are active and how they change during a conversation.

Script API (`p` object)

`p.play(response)` Makes Alan say `response`. Can be string, array of strings, or SSML.	`p.then(handler)` Sets up a follow-up intent handler for the next turn.
`p.setContext(contextName)` Sets the active context. `contextName` can be a string or `''` to clear.	`p.resolve()` Indicates the intent handling is complete for this turn.
`p.repl(script)` Dynamically adds new script code. Use with caution.	`p.callClient(methodName, params)` Calls a method on the client side via the `onCommand` handler.
`p.visual(visualState)` Updates the visual state object on the client side.	`p.nlu.text` The raw text recognized by the ASR/NLU engine.
`p.nlu.tokens` An array of recognized tokens.	`p.nlu.intent` The name of the matched intent.
`p.nlu.entities` An object containing recognized entities and their values.	`p.nlu.slots` An object containing recognized slots and their values.
`p.userData` An object to store and retrieve user-specific data across turns and sessions. Persists across sessions if enabled.	`p.state` Similar to `p.userData` but typically used for temporary state within a session. Does not persist across sessions.
`p.random(arr)` Selects a random element from the array `arr`.	`p.log(message)` Logs a message to the Alan AI Studio Debugger console.

Client API & Handlers

Client-Side Integration (Web/Mobile)

Initializing Alan Button

```
import alanBtn from '@alan-ai/alan-sdk-web';

alanBtn({
    key: 'YOUR_ALAN_KEY',
    onCommand: function(commandData) {
        // Handle commands from the script
    },
    onEvent: function(event) {
        // Handle button and conversation events
    },
    onButtonState: function(state) {
        // Handle button state changes (listening, processing, idle)
    },
    onVisualState: function(visualState) {
        // Handle visual state updates from the script
    },
    onConnectionStatus: function(status) {
        // Handle connection status changes
    }
});
```

Replace `'YOUR_ALAN_KEY'` with your actual SDK key from Alan AI Studio.

Key Handlers

- `onCommand`: Receives data sent from the script using `p.callClient(methodName, params)`. `commandData` object includes `command` (methodName) and other properties sent from script.
- `onVisualState`: Receives the object sent from the script using `p.visual(visualState)`. Use this to update your UI based on the conversation state.
- `onEvent`: Catches various events like `voice:start`, `voice:stop`, `recognizer:start`, `recognizer:end`, `script:loaded`, etc.
- `onButtonState`: Notifies when the button changes state (e.g., becomes active/listening, processing, idle).

Sending Commands to Script

Use the `callClient` method of the Alan button instance:

```
const alanBtnInstance = alanBtn({...});

// Later, to send a command:
alanBtnInstance.callClient('start_game');

// In your Alan script, you'll have:
intent({ command: 'start_game' }, p => {
    p.play('Starting the game!');
});
```

This is useful for triggering script logic from UI actions.

Setting Visual State from Client

While primarily set *from* the script using `p.visual()`, you can also update the visual state from the client if needed (though less common):

```
alanBtnInstance.setVisualState({ page: 'homepage', userStatus: 'loggedIn' });
```

This updates the `visual` property of the `p` object in the script for the *next* turn.

Sending Text to Alan

Allows sending text to Alan as if the user spoke it:

```
alanBtnInstance.sendText('what is the weather?');
```

Useful for initial prompts or integrating with chat interfaces.

Activating/Deactivating Alan

Control the button's listening state:

```
alanBtnInstance.activate(); // Start listening
alanBtnInstance.deactivate(); // Stop listening
alanBtnInstance.playText('Hello!'); // Make Alan speak without activating listening
```

Working with Visual State

The `visualState` object received in `onVisualState` should mirror your UI state. Alan script can then query this state to make decisions:

```
// Script side:
intent('go back', p => {
    if (p.visual.screen === 'details') {
        p.play('Going back to list.');
        p.visual({ screen: 'list' });
    } else {
        p.play('I can't go back from here.');
    }
});
```

Best Practice: Design your visual state carefully to represent relevant UI information the script might need.

Troubleshooting Client Integration

- Check the network requests in your browser/app's developer tools to see if the connection to Alan AI is successful.
- Ensure your SDK key is correct.
- Use `alanBtn().remove();` during component unmount/cleanup in frameworks like React/Vue to prevent memory leaks.

Advanced Topics & Best Practices

Global Variables & Data Persistence

Global Variables

Declare variables outside of `intent` blocks but inside the main script function:

```
let order = [];

intent('add $(item P:item~)', p => {
    order.push(p.item.value);
    p.play(`${p.item.value} added.`);
});

intent('what is in my order', p => {
    p.play('You have ' + order.join(', ') + ' in your order.');
});
```

Global variables persist throughout the script's lifecycle on the server.

User Data (`p.userData`)

Use `p.userData` to store information specific to the *current user*. This data can persist across sessions if enabled in the project settings.

```
intent('set my name to $(name P:name~)', p => {
    p.userData.userName = p.name.value;
    p.play('Got it, ' + p.userData.userName);
});

intent('what is my name', p => {
    if (p.userData.userName) {
        p.play('Your name is ' +
p.userData.userName);
    } else {
        p.play('I don't know your name yet.');
    }
});
```

Session State (`p.state`)

Use `p.state` for temporary data relevant only to the *current conversation session*.

```
intent('start checkout', p => {
    p.state.checkoutStep = 1;
    p.play('Okay, starting checkout. Step 1: confirm address.');
});

intent('confirm address', p => {
    if (p.state.checkoutStep === 1) {
        p.state.checkoutStep = 2;
        p.play('Address confirmed. Step 2: payment.');
    } else {
        p.play('We are not at the address step.');
    }
});
```

`p.state` is reset when the session ends (e.g., user closes the app/browser tab or after a period of inactivity).

Choosing Data Storage

- **Global Variables:** For data needed across *all* user interactions and sessions (e.g., configuration, API keys).
- **`p.userData` :** For data specific to a *user* that should persist across sessions (e.g., preferences, profile info, shopping cart).
- **`p.state` :** For data specific to the *current conversation flow* or session (e.g., current step in a multi-turn process, temporary flags).

Working with External APIs

Use `p.fetch()` or standard Node.js modules (like `axios` , `node-fetch`) to make HTTP requests from your script to external services.

```
intent('what is the weather in $(city P:city)', async p => {
    const city = p.city.value;
    const apiKey = project.apiKeys.weather;
    const url = `https://api.weatherapi.com/v1/current.json?
key=${apiKey}&q=${city}`;

    try {
        const response = await p.fetch(url);
        const data = await response.json();
        const tempC = data.current.temp_c;
        p.play(`The current temperature in ${city} is ${tempC} degrees
Celsius.`);
    } catch (error) {
        p.play('Sorry, I could not get the weather.');
        p.log(error.message); // Log error for debugging
    }
});
```

Remember to configure API keys securely in project settings.

Asynchronous Operations

Use `async/await` with `p.fetch` or other asynchronous functions to avoid blocking the script execution.

The `p.resolve()` call is important after asynchronous operations to signal that the intent handling is complete *after* the async work finishes.

Debugging Tips

- Use `p.log('message')` to print values and execution flow to the Debugger console.
- Use the 'Debugger' tab in Alan AI Studio to see user input, matched intents, recognized entities/slots, `p.log` messages, and responses.
- Use the 'Contexts' tab to track context changes.
- Use the 'Visual State' tab to see the current visual state sent to the client.
- Check 'History' to review past interactions.
- If using `p.fetch` , check the 'External Calls' tab.

Error Handling

Wrap API calls and other potentially failing operations in `try...catch` blocks to provide graceful responses to the user and log errors for debugging.

Best Practice: Modularize Script

Break down large scripts into smaller, manageable functions or files if possible (using `require` or imports if your environment supports it, or just helper functions within the main script file). This improves readability and maintenance.

Best Practice: Keep Responses Concise

Alan AI is best for short, direct interactions. Avoid long monologues. Use the visual interface for displaying detailed information.

Best Practice: Progressive Disclosure Don't ask for too much information at once. Use follow-ups and contexts to guide the user through gathering necessary details step-by-step.	**Best Practice: Provide Help/Examples** Include intents for 'help' or 'what can I say/do' that provide examples of valid commands, especially within specific contexts.
Best Practice: Multimodal Design Always consider the visual feedback alongside the voice response. Use `p.play()` for audible cues and `p.visual()` for UI updates.	**Best Practice: Testing Edge Cases** Test how your voice assistant handles unexpected inputs, misrecognitions, and errors from external services.