Week 6 Facilitator Hand-Outs

Validated, paste-ready prompts for every “switch to editor” moment in the Course 6 deck. Each block has been run end-to-end against the `heywood-inventory` (https://github.com/jeranaias/heywood-tbs) reference repository — pasting it into a fresh chat with `my-staff-app/` as the working directory produces code shaped like the matching reference file.

> **How to use this page.** Keep it open in a second browser tab the entire workshop. Each “switch to editor” slide in the Week 6 deck has a matching block (or three) below — click *Copy*, then paste into the AI chat that’s sharing the screen. The slide-number pill on each block tells you exactly which deck slide it pairs with.
>
> **The 3-minute rule.** Run a prompt. If the output doesn’t work in 3 minutes, paste the error back into the same chat. Don’t manually debug AI code — that’s the discipline you’re modelling all day.
>
> **Print fallback.** If you lose your second screen, this page prints to clean pages with the copy buttons hidden — use `Ctrl`+`P` / `⌘`+`P`.

Module 2 · Environment setup SLIDE 13 CUE

Two blocks. Paste the install prompt first — the AI walks each student through Go, Node.js, Git, and VS Code one at a time and pauses for the version output. Then, once the four versions print, paste the scaffold commands so the project skeleton lands in a single shell paste.

SLIDE 13 · MODULE 2 INSTALL

Install + verify (paste into the AI in the terminal)

Cue line for the room: “Prompt is in chat. Paste it as-is, but replace the OS placeholder with your actual OS. The AI will pause after each install — don’t skip the version checks.”

```
I'm on <Windows / macOS / Ubuntu>. Walk me through installing the latest stable versions of Go, Node.js (LTS), Git, and VS Code, in that order. After each install, give me the one-liner that prints the version. Stop after each step and wait for me to paste the version output back before moving on.
```

SLIDE 13 · MODULE 2 SCAFFOLD

Scaffold the project skeleton (run in a fresh shell)

Cue line for the room: “Once all four versions print, run these five commands. This is shell, not chat. The directory is the runway — we light it up in Module 3.”

```
mkdir my-staff-app && cd my-staff-app
go mod init my-staff-app
mkdir -p cmd/server internal data
npm create vite@latest web -- --template react-ts
cd web && npm install && cd ..
```

Module 3 · Backend from scratch SLIDE 17 CUE

Three prompts in sequence. After each one: run the server, hit it with curl, paste any error back to the AI before debugging. Don’t paste prompt 2 until prompt 1’s output is printing on `localhost:8080`.

SLIDE 17 · MODULE 3 PROMPT 1

Single-file server with health endpoint

Cue line for the room: “Paste this. Save the file the AI gives you to `cmd/server/main.go`. Then run the `go run` command and the curl — both should succeed before you touch prompt 2.”

```
I'm building a Go HTTP server in `cmd/server/main.go`. Use only the standard
library (net/http, flag, log, time). Add:

  - A `-port` string flag (default "8080") and a `-dev` bool flag (default
false).
  - A `GET /api/v1/health` handler that returns JSON
`{"status":"ok","timestamp":"..."}`.
  - Start the server with reasonable read/write timeouts.
  - Log the listen address and dev flag at startup.

Give me the entire file as one block I can paste, then the exact `go run`
command to start it and the curl to verify.
```

SLIDE 17 · MODULE 3 PROMPT 2

Extract the router and add helpers

Cue line for the room: “Same chat. Paste this. The AI should give you both files in full — replace what you have. Re-run, re-curl, then we go to prompt 3.”

```
Refactor `cmd/server/main.go` so route registration lives in a new file
`internal/api/router.go`. The new file should:

  - Export a `SetupRouter() *http.ServeMux` function that takes any deps it
    needs in a `Deps` struct.
  - Export `writeJSON(w http.ResponseWriter, status int, body any)` and
    `writeError(w http.ResponseWriter, status int, msg string)` helpers.
  - Move the health handler to use `writeJSON`.

In `cmd/server/main.go`, import the new package, build a `Deps`, call
`SetupRouter`, and pass the result to `http.Server`. Show me both files
in full.
```

SLIDE 17 · MODULE 3 PROMPT 3

Hardcoded items endpoint

Cue line for the room: “Last backend prompt. Five items, military-flavoured. The point is to feel the loop — in Module 4 we’ll replace this hardcoded array with a real store.”

```
Add a `GET /api/v1/items` endpoint to `internal/api/router.go` that returns a
hardcoded array of five military-flavored inventory items. Each item has:
id (int), title, status ("open"/"in_progress"/"blocked"/"done"),
priority ("low"/"medium"/"high"/"critical"), assigneeId, notes,
createdAt, updatedAt.

Use `writeJSON`. Hit it with curl after starting the server.
```

Module 4 · Data layer SLIDE 21 CUE

Three prompts. The first defines the contract (interface + JSON-backed implementation), the second seeds 20 realistic items, the third rewires the router to read from the store and adds `/items/{id}`.

SLIDE 21 · MODULE 4 PROMPT 1 DataStore interface + JSONStore

Cue line for the room: “Two files in one prompt. The interface in `store.go`, the JSON implementation in `json_store.go`. Atomic-rename writes — that’s the durability move.”

```
Create `internal/data/store.go` defining:
  - An `Item` struct with id, title, status, priority, assigneeId, notes,
    createdAt, updatedAt (camelCase JSON tags, time.Time for timestamps).
  - A `ListFilter` struct with Status, Priority, AssigneeID, Query.
  - A `Store` interface with List, Get, Create, Update, Delete, Stats, Close.
  - A `Stats` struct with Total, ByStatus, ByPriority, Recent (last 5 items).
  - Const Status* and Priority* values; an exported `ErrNotFound`.

Then create `internal/data/json_store.go` implementing `Store` against a JSON
file. Read the file on `NewJSONStore(path)`, write the file on every mutation,
guard with sync.RWMutex. Use atomic rename (write to .tmp then rename) for
durability.

Give me both files in full.
```

SLIDE 21 · MODULE 4 PROMPT 2 Seed data — 20 realistic items

Cue line for the room: “Save the AI’s output to `data/items.json`. Open the file in VS Code — eyeball the variety. If every item is ‘open / high’, ask the AI for more spread.”

```
Generate `data/items.json` containing 20 realistic military-flavored
inventory items as a JSON array using the schema in store.go. Mix all four
statuses and all four priorities. Distribute assigneeId across "admin",
"staff", and "user". Use timestamps in April 2026.
```

SLIDE 21 · MODULE 4 PROMPT 3

Rewire the items endpoint and add `/items/{id}`

Cue line for the room: “Same chat. The AI should hand back both files in full. After this paste, `curl /api/v1/items/3` must return one item; `curl /api/v1/items/999` must 404.”

```
Update `internal/api/router.go`:
  - Add `Store data.Store` to the `Deps` struct.
  - Replace the hardcoded `/api/v1/items` handler with one that calls
    `Store.List({})` and returns the result.
  - Add `GET /api/v1/items/{id}` (use Go 1.22 path values + strconv.Atoi)
    that calls `Store.Get(id)` and 404s on `data.ErrNotFound`.

Update `cmd/server/main.go` to construct a
`data.NewJSONStore("data/items.json")`
and pass it via Deps. `defer store.Close()`.

Show me both files in full.
```

Module 5 · Frontend from scratch SLIDE 26 CUE

One big prompt. Tailwind config + Vite proxy + typed API client + a Tailwind table that calls the backend on mount. This is the moment the data becomes visible to a human — have one student share their screen at the debrief.

SLIDE 26 · MODULE 5 SINGLE PROMPT

Tailwind + proxy + items table

Cue line for the room: “One prompt does the whole thing — config, client, table. Most failures here are PostCSS / Tailwind config or a missing proxy block. Both are 30-second AI fixes once you paste the error.”

```
In `web/`, set up Tailwind CSS (init config, content globs, autoprefixer,
postcss.config.cjs). Add a `server.proxy` block to `vite.config.ts` that
proxies `/api` to `http://localhost:8080` with `changeOrigin: true`.

Create `web/src/api/client.ts` with a typed `Item` interface matching the Go
struct, and an `api.listItems()` function using fetch.

Replace `web/src/App.tsx` with a component that calls `api.listItems()` on
mount and renders the items as a Tailwind-styled table with columns: ID,
Title, Status, Priority, Assignee. Color-code status and priority chips.
Use fetch with `credentials: "include"`.

Run both servers and show me the page at localhost:5173.
```

Module 6 · Pages & navigation

SLIDE 30 CUE

Three prompts. Routing + sidebar layout, then the item detail page, then the dashboard with stat cards. By the end of this module the React app looks like a product, not a demo.

SLIDE 30 · MODULE 6 PROMPT 1

Routing + sidebar layout

Cue line for the room: “Tailwind defaults only. Don’t let the AI pick fonts or icons — that’s a 20-minute rabbit hole. We want the structure first.”

```
Install `react-router-dom`. Wrap `<App />` in `<BrowserRouter>` in
`main.tsx`. Create `web/src/layouts/Layout.tsx` with:

  - A 240px dark sidebar (`bg-[#1a1f36]`) on the left.
  - Sidebar links via `NavLink`: Dashboard, Items, Chat (Settings will come
    later, conditionally).
  - Active-link styling using NavLink's className-as-function.
  - A header bar across the right column.
  - `<Outlet />` for nested routes.

Then update `App.tsx` to declare routes: index → /dashboard, /dashboard,
/items, /items/:id, /chat, /settings. Use Tailwind defaults; do not pick
fonts or icons.
```

SLIDE 30 · MODULE 6 PROMPT 2

Item detail page

Cue line for the room: “Click any title in the table — you should land on the detail page. Back link should return you to the list. If it doesn’t, paste the error.”

```
Create `web/src/pages/ItemDetail.tsx` rendered at `/items/:id`. It should:

  - useParams to read the id.
  - Fetch via `api.getItem(id)` (add this method to client.ts).
  - Show the title, ID, status chip, priority chip, assignee, dates, and notes.
  - Have a "← Back to items" link to `/items`.

In the items table, make every title cell a `<Link to={`/items/${id}`}>`.
```

SLIDE 30 · MODULE 6 PROMPT 3

Dashboard with stats + recent activity

Cue line for the room: “Adds a backend stats endpoint, then a frontend dashboard. Four cards plus a recent-activity list. The Total card should match the count in the items table — if not, look at the filter.”

```
Add a `GET /api/v1/stats` endpoint on the backend that calls Store.Stats().
Add `api.stats()` to client.ts.

Create `web/src/pages/Dashboard.tsx` with four stat cards (Total, Open,
In progress, Blocked) and a "Recent activity" section listing the 5 recent
items with status and priority chips, each linking to its detail page.
Use Tailwind defaults.
```

Module 7 · AI chat integration

SLIDE 34 CUE

Three prompts. Chat service against OpenAI, the `lookup_items` tool definition, then the React chat page with markdown. End by asking “what are my high-priority items?” — the answer should cite real items from the store and the server log should show a tool call.

SLIDE 34 · MODULE 7 PROMPT 1

Chat service against OpenAI

Cue line for the room: “Confirm everyone has `OPENAI_API_KEY` exported in their shell before pasting this. The 4-iteration loop is what lets multi-step tool calls finish.”

```
Create `internal/ai/chat.go` exposing:

  type ChatService struct { ... }
  func NewChatService(apiKey string, store data.Store) *ChatService
  func (c *ChatService) Reply(ctx context.Context, userMessage, role string)
(string, error)

Use net/http to POST to `https://api.openai.com/v1/chat/completions` with
model "gpt-4o". The system prompt instructs the AI to call tools rather than
guess. Loop up to 4 times so multi-step tool calls terminate. Return the
final assistant message content as markdown-ready text.

Wire it into the router as `POST /api/v1/chat` taking `{"message":"..."}`
and returning `{"reply":"..."}`. Init the service in main.go from
`os.Getenv("OPENAI_API_KEY")`.
```

SLIDE 34 · MODULE 7 PROMPT 2

`lookup_items` tool definition

Cue line for the room: “Same file. The role-based filter inside `runTool` is what makes RBAC actually reach the chat — we’ll prove it works in Module 8.”

```
In `internal/ai/chat.go`, define the OpenAI tools array with one tool:

  Name: "lookup_items"
  Description: "Query the inventory items database. Use this any time the
    user asks about counts, statuses, priorities, or specific items.
    Returns matching items and the total count."
  Parameters: object with optional `status` (enum:
open/in_progress/blocked/done),
    `priority` (enum: low/medium/high/critical), `query` (string substring).

Implement `runTool(call, role)` that parses the function arguments, builds a
`data.ListFilter` (apply role-based filtering: if role == "user", set
AssigneeID to "user"), calls `store.List(filter)`, and returns a JSON string
of `{ count, items: [...] }` to send back to OpenAI.
```

SLIDE 34 · MODULE 7 PROMPT 3

Chat page with markdown rendering

Cue line for the room: “End the module by asking *‘what are my high-priority items?’* — you should see real item titles in the answer and a tool call in the server log. That’s the architecture diagram going from solid to lit.”

```
Install react-markdown. Create `web/src/pages/ChatPage.tsx` with:

  - A scrollable transcript area; assistant turns rendered through
    <ReactMarkdown>, user turns plain.
  - An input + Send button at the bottom.
  - "Thinking…" placeholder while waiting on the response.
  - An error banner above the input on failure.

Add `api.chat(message)` to client.ts. End by asking "what are my
high-priority items?" — confirm the answer cites real items from the
database, and watch the server log a tool call.
```

Module 8 · Auth & middleware

SLIDE 39 CUE

Three prompts. The middleware package, the auth endpoints + role picker, then role-based filtering across every items handler and the chat tool. The verification is visible: switch to User and the items list, dashboard counts, and chat answers all narrow together.

SLIDE 39 · MODULE 8 PROMPT 1

Middleware package

Cue line for the room: “The Chain function order matters. Auth needs to run after CORS so the cookie is read on the actual request, not the preflight.”

```
Create `internal/middleware/middleware.go` exporting:

  - `Middleware = func(http.Handler) http.Handler`
  - `Chain(h http.Handler, mws ...Middleware) http.Handler`
  - `RequestLogger(dev bool)`, `SecurityHeaders()`, `CORS(dev bool)` (allow
    http://localhost:5173 with credentials when dev=true), `Auth()` (reads a
    "heywood_role" cookie, defaults to "user", stamps the role into request
    context).
  - A `RoleFrom(*http.Request) string` helper.
  - Const RoleAdmin/RoleStaff/RoleUser and CookieRole.

In main.go, wrap the mux with `middleware.Chain(mux, RequestLogger,
SecurityHeaders,
CORS, Auth)`.
```

SLIDE 39 · MODULE 8 PROMPT 2

Auth endpoints + frontend role picker

Cue line for the room: “After this lands, the role picker appears in the header. Switch roles — the page reloads and the Settings link appears for Admin only.”

```
Add two routes:
  - `GET /api/v1/auth/me` returns `{ role, canAdmin, canCreate }`.
  - `POST /api/v1/auth/switch` takes `{"role":"admin|staff|user"}`, validates,
    sets the heywood_role cookie (Path=/, SameSite=Lax, MaxAge=7d).

On the frontend, add `api.me()` and `api.switchRole(role)` to client.ts.
Create `web/src/components/RolePicker.tsx` — a select bound to the current
role; on change, call switchRole then `window.location.reload()`. Render it
in the Layout header. Conditionally render the Settings sidebar link when
`me.canAdmin` is true.
```

SLIDE 39 · MODULE 8 PROMPT 3

Role-based filtering across every endpoint

Cue line for the room: “The verification is the demo. Switch to User — items list, dashboard counts, and chat answers should all narrow to the User’s items together. If one of them doesn’t, you have a leak.”

```
In every items handler, call `middleware.RoleFrom(r)`:
  - List: if role == "user", force ListFilter.AssigneeID = "user".
  - Get: if role == "user" and item.AssigneeID != "user", return 404 (don't
    leak existence).
  - Create: forbid User entirely (403).
  - Update: User can only update items where assigneeId == "user".
  - Delete: Admin only.
  - Stats: if role == "user", build the dashboard payload from
    `Store.List({AssigneeID: "user"})` so the four stat cards and the
    "Recent activity" feed match the table the User actually sees. Admins
    keep the existing `Store.Stats()` fast path.

In ai/chat.go's runTool, apply the same filter so the chat respects RBAC.
Verify by switching to User in the role picker and watching the items list,
the dashboard counts, and the chat answers all narrow together.
```

Module 9 · External integrations

SLIDE 43 CUE

Two paths — pick one for the room. **Path A (SQLite)** is the default; it proves the `Store` interface earns its keep. **Path B (Microsoft Graph)** is for rooms that have the tenant-side env vars ready and want to wire calendar / mail. If the room is split, run Path A live and post Path B to chat for self-study.

SLIDE 43 · MODULE 9 PATH A

SQLite store (default for the room)

Cue line for the room: “Same `Store` interface, new implementation behind a build tag. Build with `-tags sqlite`, restart, POST a new item, restart again, GET to confirm it persisted. That’s the Liskov win.”

```
Create `internal/data/sqlite_store.go` (build tag `sqlite`) implementing the
same Store interface against `database/sql` + the pure-Go `modernc.org/sqlite`
driver. Auto-create the `items` table on startup with indexes on status,
priority, assignee_id. Use parameterized queries.

Add a stub `sqlite_stub.go` (build tag `!sqlite`) so the default build still
compiles when modernc.org/sqlite is absent. Stub returns "sqlite store not
compiled in" from NewSQLiteStore.

Add a `-db` flag to main.go (default "json", accepts "sqlite"). Build with
`go get modernc.org/sqlite && go build -tags sqlite ./cmd/server`. Run with
`./server -db sqlite -dev`. POST a new item, restart, GET to confirm
persistence.
```

SLIDE 43 · MODULE 9 PATH B

Microsoft Graph (calendar + mail)

Cue line for the room: “This only fires if the four `GRAPH_*` env vars are set. The graceful nil-from-`New()` pattern is what keeps the dev-laptop build working when the integration isn’t available.”

```
Create `internal/integrations/graph.go`:
  - Read GRAPH_TENANT_ID, GRAPH_CLIENT_ID, GRAPH_CLIENT_SECRET, GRAPH_USER
    from env. Return nil from `New()` when env vars are absent.
  - Support GRAPH_CLOUD = "commercial" (default) | "gcchigh" with the right
    login + graph base URLs for each.
  - OAuth2 client-credentials flow with token caching (refresh 1 minute
    before expiry).
  - Methods: `CalendarToday(ctx) ([]CalendarEvent, error)` and
    `MailSummary(ctx) (MailSummary, error)`.

In main.go, call integrations.New(); if non-nil, pass to Deps.Graph. In the
router, register `/calendar/today` and `/mail/summary` only when Graph != nil.
```

Module 10 · Docker & deployment

SLIDE 47 CUE

The capstone paste. One prompt produces the SPA fallback, the multi-stage Dockerfile, and the `.dockerignore`. The optional Azure block ships it to Azure Container Apps in four commands — only paste it if the room has a subscription wired up.

SLIDE 47 · MODULE 10 SINGLE PROMPT

Multi-stage Dockerfile + SPA fallback

Cue line for the room: “After the build succeeds, kill your local dev servers, then reload `localhost:8080`. The whole app should still work — that proves the container is self-contained.”

```
Add a SPA fallback to `internal/api/router.go`: when `Deps.WebDir` is set,
register a "/" handler that serves files from WebDir if they exist on disk,
otherwise serves WebDir/index.html. Refuse to serve anything under /api/.

Add a `-web` flag to main.go (default "web/dist") wired into Deps.WebDir.

Create a multi-stage `Dockerfile` and `.dockerignore`:
  - Stage 1 `node:20-alpine`: COPY web/, npm install, npm run build.
  - Stage 2 `golang:1.22-alpine`: COPY ., go build with CGO_ENABLED=0.
  - Stage 3 `alpine:3.20`: ca-certificates, non-root user, COPY the binary,
    the dist/ from stage 1, and data/ from stage 2. EXPOSE 8080.
    ENTRYPOINT ["/app/server", "-port=8080", "-web=/app/web/dist", "-
data=/app/data/items.json"].

Show me both files. Then:

  docker build -t my-staff-app .
  docker run -p 8080:8080 -e OPENAI_API_KEY=$OPENAI_API_KEY my-staff-app

Open localhost:8080 — the whole app should work from one container. Stop
your local dev servers and reload to prove it.
```

SLIDE 47 · MODULE 10 OPTIONAL

Azure Container Apps (optional, paste only if the room has a subscription)

Cue line for the room: “Replace the four placeholders before pasting. `az login` first, in a real terminal, not the AI chat. The container apps URL it spits back is the live, public app.”

```
az login
az acr build -r <your-acr> -t my-staff-app:latest .
az containerapp create \
  --name my-staff-app \
  --resource-group <rg> \
  --environment <env> \
  --image <your-acr>.azurecr.io/my-staff-app:latest \
  --target-port 8080 \
  --ingress external \
  --secrets openai-key=$OPENAI_API_KEY \
  --env-vars OPENAI_API_KEY=secretref:openai-key
```