1. Why Persist?

React state lives in memory. Reload? It's gone. Persistence lets data survive:

- Page refreshes
- Browser restarts
- (Sometimes) Multiple tabs

Typical categories:

- Ephemeral UI state (input focus, open modal) → Do NOT persist
- Domain data (survey answers) → Yes, persist
- Derived/computed state (filters, sorted lists) → Recompute, don't persist

2. Local vs. Remote Storage

Two fundamental approaches:

Local Storage (Client-Side)

- Data stays on user's device
- Fast, works offline
- Device-specific (no sync)

Remote Storage (Server-Side)

- Data lives on servers
- Syncs across devices/users
- Requires network connection

3. Local Storage with React

An alternative to `useState` that automatically persists to browser:

```
import { useLocalStorage } from 'usehooks-ts';

// Instead of useState:
const [count, setCount] = useState(0);

// Use useLocalStorage:
const [count, setCount] = useLocalStorage('count', 0);
```

Key Differences:

- Takes a storage key as first parameter
- Automatically saves to localStorage on every update
- **Hydrates** from localStorage on component mount
    - i.e., initializes state from stored value if present

4. Example: A Mix of Ephemeral and Persistent State

```
function SurveyPage() {
  // PERSISTENT: User's fruit selection survives reloads
  const [selectedFruit, setSelectedFruit] = useLocalStorage<string | null>('selectedFruit', null);

  // PERSISTENT: User's prediction survives reloads
  const [prediction, setPrediction] = useLocalStorage<number | null>(
    selectedFruit ? `prediction_${selectedFruit}` : 'prediction_',
    null
  );

  // EPHEMERAL: Input field state - doesn't need to persist
  const [predictionInput, setPredictionInput] = useState('');

  // Hydrate ephemeral state from persistent state if present
  useEffect(() => {
    if (prediction !== null) {
      setPredictionInput(String(prediction));
    } else {
      setPredictionInput('');
    }
  }, [selectedFruit]);
}
```

Key Pattern: Ephemeral UI state gets **hydrated** from persistent domain state.

5. Local Storage Key Names

Storage Keys: Strings that identify your data

```
useLocalStorage('userSettings', {});     // Simple key
useLocalStorage(`cart_${userId}`, []);   // Dynamic key
```

6. Local Storage Data Types

Only strings are stored; not all objects can be converted to strings well (serialized)

```
// ✅ These work
useLocalStorage('name', 'John');
useLocalStorage('age', 25);
useLocalStorage('user', { name: 'John', age: 25 });

// ❌ These don't serialize well
useLocalStorage('callback', () => {});   // Functions lost
useLocalStorage('date', new Date());     // Becomes string
```

7. Storage Limits

- Browser limits storage to ~5-10MB per domain (per website)
  - `example.com` gets 5-10MB total
  - `github.com` gets its own separate 5-10MB
  - All localStorage keys for a site share this limit
- Exceeding limit throws `QuotaExceededError`
- Clear old data or use a different library like IndexedDB for large datasets

8. Common Issues to Mention to Copilot:

- "My data disappears on reload" → Check storage key spelling
- "Getting 'undefined' instead of my object" → Check JSON parsing
- "Storage quota exceeded" → Need to clear old data or use a different library like IndexedDB for large datasets

Types of Remote Storage Architectures

- **Key-Value Store**: Simple, fast, stores data as key → value pairs (e.g., Redis, DynamoDB)
- **Document Store**: Stores JSON-like documents, flexible schema (e.g., MongoDB, Firebase)
- **Relational (SQL) Database**: Structured tables, relationships, powerful queries (e.g., PostgreSQL, MySQL, SQLite)
- **File/Object Storage**: Stores files or blobs, not structured for queries (e.g., AWS S3, Google Drive)

For our survey app, we'll focus on the SQL (relational) option.

SQL Databases: Tables and Joins

- A SQL database is a set of tables (like spreadsheets)
- Each table has columns (fields) and rows (records)
- Tables are related by shared columns (keys)
- You can combine data from multiple tables using a **JOIN**
    - like we saw with movies and ratings in week4
- A query language (SQL) is used to interact with the data
    - filter, sort, aggregate, join, etc.
    - analogous to same operations we did in python with pandas

Designing a Database for a Survey App

- Goals:
  - Support multiple survey questions
  - Record user answers
  - Display feedback based on answers received so far
- Discussion question: What tables should we create? How are they related?

Step 1: Identify the Entities

Let's assume we have only one survey for simplicity.

- **Question**: One question in the survey, like favorite fruit
- **Answer-Options**: Possible choices for a survey question
- **User**: (Optional) Who is answering?
- **Answer**: A user's response to a question

Design Choice: Normalization

Normalization is the process of organizing data to minimize redundancy. In our case, we want to ensure that:

- Questions and their answer options are stored separately
- User information is not duplicated across answers

This leads to a more efficient database design.

Ask your copilot to help you design a normalized schema.

Example: Should Answer Options Be Columns, or a Separate Table?

We could make the answer options be columns in the questions table:

id	question_text	option1	option2	option3
1	"Favorite fruit?"	"Apple"	"Banana"	"Cherry"
2	"How many per week?"	"1-3"	"4-6"	"7+"

But this is inflexible:

- What if a question has more or fewer options?
- What if we want to add metadata to options (e.g., is_correct)?

The Separate Table Approach

`questions` table:

id	text
1	"Favorite fruit?"
2	"How many per week?"

`answer_options` table:

id	question_id	text
1	1	"Apple"
2	1	"Banana"
3	1	"Cherry"
4	2	"1-3"

A Schema for Our Survey App

Table	Fields/columns
questions	id, text
answer_options	id, question_id, text
users	id, name, email (optional)
answers	id, user_id, question_id, answer_id, free_answer

Step 3: Relationships

- A question has two or more answer options
- Each answer links a user to a question and an answer_option

answers

id	user_id	question_id	answer_id	free_answer
1	42	1	2	NULL
2	42	2	NULL	"5"

DB Operations Overview

- **Create**: Insert new records (e.g., new questions, answers)
- **Read**: Query existing records (e.g., get all answers for a question)
- **Update**: Modify existing records (e.g., change an answer)
- **Delete**: Remove records (e.g., delete a user's answer)

Designing API Endpoints

We don't let users interact with the database directly, because:

- Security: Prevent unauthorized operations
- Validation: Ensure data integrity and enforce business rules
    - For example, don't add an answer with an invalid answer_option

Instead, we make one API endpoint for each operation the app needs.

- Get all survey questions and answer_options
- Get all existing answers for the current user
- Add or update one answer
- Get answer counts for a question

Using the Database in Our Survey App

- We will use SQLite for simplicity (file-based, only for small databases)
- See the `backend/` folder for the full implementation
- See `schema.sql` for the database schema
- See `seed.sql` for example data
- See `server.cjs` for the API endpoints
- To initialize and run, see the README in that folder

Danger Zones

- User accesses or changes data they shouldn't (security and privacy)
  - We will cover authentication/authorization in a future week
- User submits invalid data (e.g., wrong answer type)
- User submits malformed data (e.g., SQL injection)
  - If you're not careful, this can cause arbitrary code execution on the server!
  - Probably the most common security vulnerability in web apps

How to Avoid SQL Injection

- If user input will be included in a SQL query, **never** concatenate strings directly.
- Instead, use **parameterized queries** (also called prepared statements).
- We have added something about this to the `react_practices.md` file, but remind your copilot if needed.