Concept/Topic: *State Management*

Context and Motivation

Statelessness of interactions of interactions between service consumer/requestor (or client) and service provider (or server)[1] helps to achieve loose coupling. It makes it easier to scale out and to deploy rapidly and flexibly. Statelessness is emphasized both in SOA and REST.

Background: Why Bother About State Anyway?

State is at the heart of information technology and digitalization. Only the most trivial applications such as a mere calculator accessory do not keep any data steering their behavior. Keeping such state data speeds up request processing and supports computations. However, state data also is subject to audit and often the target of security attacks (e.g., customer contact information in CRM solutions and credit card numbers in payment systems). State hinders elasticity in clouds (dynamic workload management).

Hence, state management is a key Architecturally Significant Requirement (ASR) and stakeholder concern in information system design in particular, but also in other application genres (for instance, Internet of Things). If state is not preserved correctly, for example because it has been tampered with or because it has been updated inconsistently in two parts of a system, computations may fail or, even worse, are performed under invalid assumptions which has a negative impact on future user requests. In an enterprise information systems context, for instance, insurance rates might be too high because of wrongly dispatched claim forms or loan requests are declined due to a presumably poor credit history that is not actually true, but stored in the responsible risk management systems this way. Such data quality problems might go unnoticed for a long time.

Definition(s)

Terminology conflicts and, to some extent, confusion can be observed:

- The PhD thesis that introduced REST talks about "application state" (Fielding (2000)), but later publications and online posts by R. Fielding also talk about "resource state" and point out that these two are not the same (without going into detail).
- The PoEAA book (and the CQRS, and Event Sourcing patterns also described by M. Fowler) distinguish between "session state" and "application state" as two types of observable state[2] (Fowler (2002)).
- G. Hohpe, a co-author of the EIP book, discusses "conversation state" and more types of state in one of his ramblings (see here[3]).

Two Types of Application State: Session State vs. Resource State

Layering comes to the rescue by answering the question which layer and component(s) are responsible for owning/holding the state:

- The presentation layer holds the session state.

[1] Note that application components such as service providers and consumers and the middleware that supports them with integration capabilites (e.g., queue manager, Web server) are inherently coupled via the middleware invocation API (which can be a local or a remote one).
[2] https://martinfowler.com/bliki/ObservableState.html
[3] http://www.enterpriseintegrationpatterns.com/ramblings/20_statelessness.html

- The Business Logic Layer (BLL) and its subordinate layers (data access, persistence) deal with resource state as well as (business) conversation state.

Hence, we will use *application state* as the general term (for any server-side, tier 2/tier 3 state) and *session state* and *resource state* as the two layer-specific refinements of the general concept. Both types of state influence the quality properties of a system heavily (as already discussed in the Context and Motivation section).

Session State Management Patterns (Options)

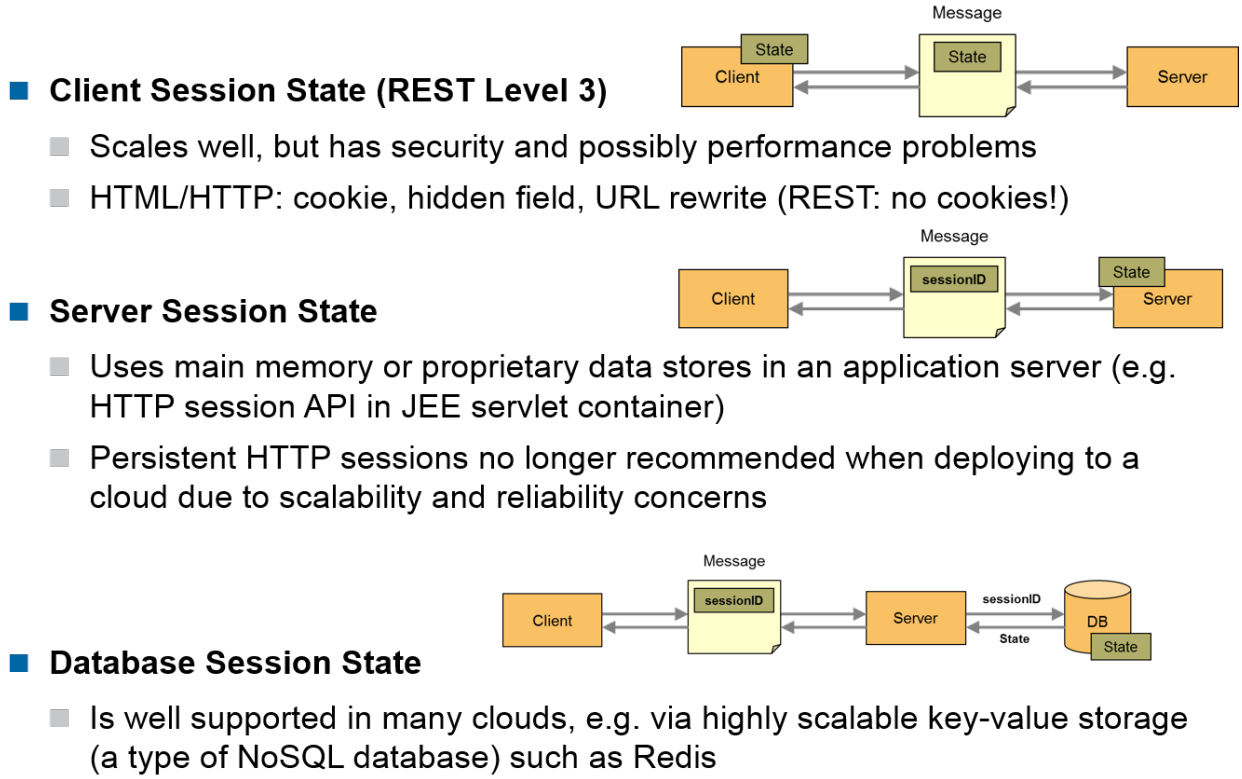

Figure 1: *Session State Management patterns in PoEAA Fowler (2002) (Reference: Frank Leymann, IAAS Stuttgart)*

Additional decision drivers and more pattern pros and cons (not yet outlined in the figure "Session State Management patterns in PoEAA") include:

- Server session state has the smallest development effort (if supported by server-side Web framework), followed by client session state and database session state. That said, database access logic is typically required anyway to implement durability requirements (persistence); hence, there are synergies.
- Server session state does not go well with clustered deployments because session stickiness and/or persistence has to be configured in the load balancer and/or sessions be made available to all cluster members (which again requires persistence). Session persistence is required in hot standby and failover scenarios; it turns the server session state into a special form of database session state (using a database that is managed and controlled by the application server and not the programmer).
- The performance impact of each of the patterns not only depends on the amount of data to be stored, but also on the deployment architecture and implementation quality (performance best practices to be adhered to); it is recommended to measure it continuously both in a controlled and in a realistic near-production environment (see related exercise in VSS that features tools such as Gatlin to do so).

State Transition vs. State Transfer

In distributed systems, terminology confusion easily occurs as several different perspectives can be taken. A closer look unveils that state management in distributed systems actually comprises two related but different architectural concerns and decisions required: *state transfer* vs. *state transition* (or: server-internal state management vs. client-server interactions; external vs. internal state; component vs. connector state).

Example(s)

As defined above, there are multiple flavors of application state: session state vs. resource state. An example of session state is the page flow in a thin client (multi-page user input, e.g. registration form). This temporary state is not interesting for the domain layer yet (so transient unless servers are clustered). Examples of resource state are order content and status, customer purchase history (persistent) in an online shop; generally speaking, all DDD entities qualify as resource state.

See coverage of state and layers in the real project that inspired the "T" case study (Zimmermann et al. (2005)) for additional examples.

Application of the Concept in Products and Projects

Usage of Concept/Topic

These days it is had to find local, centralized systems; decentralization and integration which leads to all known challenges/fallacies of distributed computing, including dealing with the consequences of the Consistency, Availability, Partition Tolerance (CAP) theorem ("two out of three ain't bad"). In distributed systems, stateful components become bottlenecks and points of failure easily; they are harder to migrate to a virtualized environment (and to move within such environments), harder to backup and restore than stateless ones, etc. State that changes often, possibly via multiple channels, is harder to cache because cache invalidation and consistency management are far away from being trivial.

State modeling and state management design and have been discussed since the very beginnings of computing; design patterns have been harvested from project experience, and technology level advice been given. For instance, the classical "Design Patterns" book by the Gang of Four features a pattern called State.

While true REST would mandate to put all application state information into links (both session state and resource state), session state management with session ids and a session database (e.g., a key-value storage or a dedicated table in an already existing relational database) is a common approach in practice.

Event sourcing (possibly supported by CQRS) can be seen as an alternate approach to state management; see separate fact sheet.

Tips and Tricks

Classic conflicts include:

- Flexibility/footprint vs. efficiency of lookups (keep little state, keep a lot of state).
- Consistency vs. availability vs. partition tolerance (CAP) tradeoff; backup capability, consistency, availability (BAC) tradeoff.
- Security (expose little) vs. efficiency (more lookups and communication needed).

Adding to the complexity of the topic are aspects such as dealing with multiple frontend channels, backend processing, transactional guarantees (in end-to-end architectures and system of systems).

A greatly simplified best practice advice, which should be handled with care, is to use client session state if performance and security requirements are light and to use database session state in call other cases; server session state should be avoided due to its impact on scalability (also see the discussion on this topic in exercise 12). One should always take the actual project requirements and context into account (functional and non-functional, requirements and constraints) when picking one of the three patterns. This activity qualifies as a recurring Architectural Decision (AD) because many thin clients on the Web require session state management (as the HTTP protocol is stateless).

More Information

Related Topics and Concepts

Service design topics such as coupling criteria and granularity.

Miscellaneous Links:

- Via IFS website Architectural Knowledge Hubs[4]

References

Fielding, Roy Thomas. 2000. "REST: Architectural Styles and the Design of Network-Based Software Architectures." Doctoral dissertation, University of California, Irvine. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Fowler, Martin. 2002. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Zimmermann, Olaf, Vadim Doubrovski, Jonas Grundler, and Kerard Hogg. 2005. "Service-Oriented Architecture and Business Process Choreography in an Order Management Scenario: Rationale, Concepts, Lessons Learned." ACM, 301–12.

[4] https://www.ifs.hsr.ch/index.php?id=13193&L=4