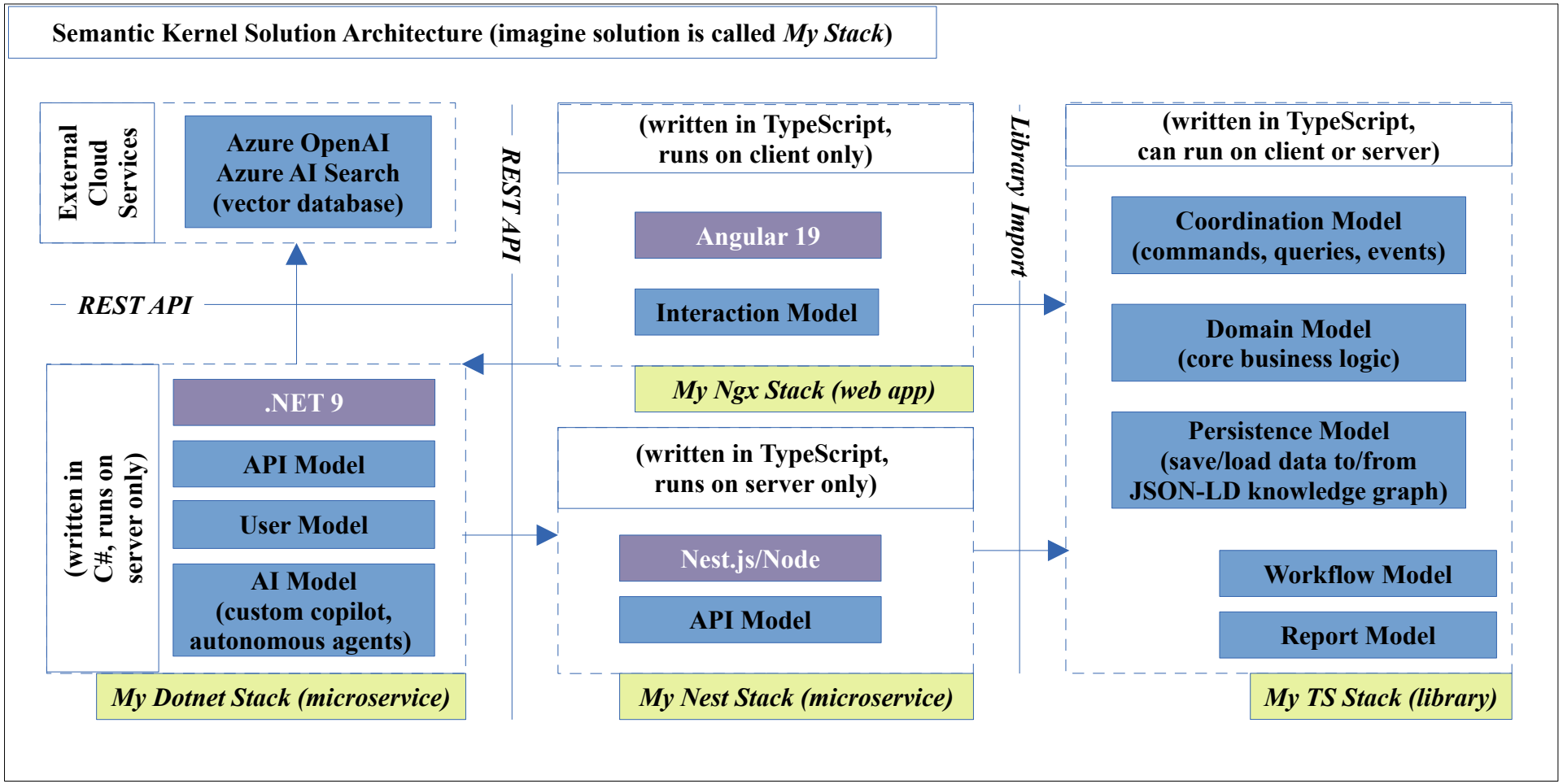
CLIPCODE™

https://clipcode.net/mentoring

Semantic Kernel Solution Architecture

Angular Super app+selection of micro apps, OpenAPI, .NET Microservices, Semantic Kernel, Azure OpenAI, Azure AI Search, Nest.js/Node Microservices



Requirements

We need to build a large-scale enterprise solution or competitive commercial product, that is meant to last 5-7 years (with regular updates) with high number of users.

We wish to use the most advanced web UI framework for building rich user experiences, which is Angular, with code written in TypeScript.

We wish to be able to use the most advanced AI framework, which is Semantic Kernel. This supports a number of languages, the furthest along being C#. We wish to use it to build powerful custom AI copilot and autonomous AI agents.

We wish to write as little code as possible (the best engineer writes the least amount of code!) – specifically we do not wish to have to re-code the same domain logic for client and server.

We wish to be able to run the domain models - the core business logic of the micro apps – on both the client and the server (so that we can **BOTH** offer the capability of running offline (PWA app) and run un-attended from the cloud (e.g. have the cloud-based AI autonomous agent, automatically responding to an event at 2am with no user present, calling into the cloud-based domain models as needed).

We wish to be able to generate reports (PDF, Word, Excel, ..) while running offline (no cloud connection) and create in the cloud (with no user present) automated end-of-month reports.

We wish to be able to execute workflows (sets of discrete actions) on the client and in the cloud.

Architectural Approach

Our key architectural choices include:

- Keep design as simple as possible
- Use of domain driven design
- As much as makes sense, have code that runs anywhere
- Use of JSON-LD knowledge graphs as data format
- Use of super app and family of micro apps
- Use of Semantic Models (User Model, AI Model, Report Model, ..) Sub-divide functionality into semantic models

Often in-experienced teams devise from the outset un-neccessarily complex architectures that simply are not needed, or not needed at present. If a solution becomes commercially very successful in future, it is likely there will be plenty of budget to have a bigger team and work on larger designs; but at the beginning when there is a smaller team and everyone is under time pressure to quickly deliver, it make a whole lot of sense to keep everything as simple as possible, to suit the immediate needs of the project now and say six months into the future (where we might have reasonably clear visibility of what is useful/needed).

The core business logic should be separated out from infrastructural and UI code, in the form of a clean domain model. To satisfy the requirements, we want this to be able to run on both the client and the server.

The data representation format should be flexible, AI-friendly and take a modern approach, hence the use of JSON-LD knowledge graphs.

Delivering the solution as a super app composed of a flexible family of micro apps offers a number of benefits.

The super app provides the superstructure within which the micro apps run and offers some shared functionality (way to sign-in/sign-out, notifications, a launcher to view available micro apps and to start one; access to content / documentation, shared search/filter and more).

The micro apps are for discrete pieces of functionality. Some can offer common functionality (User Manager, Report Manager, Tracker) and some are "premium" and are specific to a solution. The range of micro apps may well evolve over 5-7 years of the lifespan of a solution. Also some micro apps may be reused in different super apps. Micro apps can – LEGO like – be added to a super app as appropriate.

Software should be structured based on semantic models. The idea with models is that they are representations of a solution from a particular viewpoint (that are consistent with each other). A good solution delivers a range of semantic models that work together. Sample include: Domain Model, Error Model, Diagnostics Model, Interaction Model and Policy Model.

Technology Selection

For programming languages, we select two: TypeScript for UI development and any code that needs to run (unchanged) on both client and server; and C# for code that can run on server only.

For AI framework, we pick Microsoft's Semantic Kernel. This is a rapidly developing state-of-the-art AI orchestration framework with support for a range of connectors and latest ideas such as agents. The main alternative is LangChain, but SK has more of what we need and can be programmed in C#.

For UI framework, we pick Angular. For large solutions, Angular is significantly better than competitors such as React (suffers from "library churn") and Blazor (e.g. nothing in either similar to the open source Angular Material, released in lockstep with six-monthly Angular updates).

For running TypeScript-based domain models on server, we pick Nest.js (not to be confused with totally different but similarly named Next.js). A developer with experience of Angular will very quickly get up to speed with Nest.js.