Proven Architecture

Dr. Raman Ramsin

Firoozeh Abrishami, Reza ErfanArani, Taha JahaniNezhad, Alireza Moradian

Contents

مقدمه

در این سند به بررسی انواع معماری‌های نرم‌افزاری می‌پردازیم و بهترینش را با توجه به نیاز پروژه و دانش اعضای تیم برای انجام هرچه بهتر و دقیق‌تر پروژه انتخاب می‌کنیم.

معماری نرم‌افزار عبارت است از روشی که ما تصمیم می‌گیریم تا با استفاده از آن زیربنای نرم‌افزار خود را بسازیم و نرم‌افزارمان را روی این زیربنا، بنا کنیم. در کتاب Software Architecture Patterns از Mark Richards ما پنج نوع معماری که اکثریت برنامه‌نویسان از آن بهره می‌برند را می‌توانیم پیدا کنیم که در این سند ما این پنج مورد را بررسی می‌کنیم و یکی از آن‌ها را به عنوان proven architecture می‌پذیریم.

معماری‌ها

معماری لایه‌ای (Layered)

این روش احتمالا رایج‌ترین روش بین برنامه‌نویسان حال حاضر باشد. در این روش برنامه حول دیتابیس ساخته می‌شود و بسیاری از اپلیکیشن‌ها نیاز به ذخیره‌سازی دیتا دارند. بسیاری از frameworkهای برنامه‌نویسی نیز از این روش ساخته شده‌اند (مانند Java EE ،Express و...) و این باعث می‌شود که درون این چارچوب‌ها برنامه‌ها به سمت لایه‌ای نوشته شدن پیش روند.

در این معماری ما معمولا **سه لایه** داریم. لایه‌ی View، لایه‌ی Controller یا Presenter و لایه‌ی Model. برنامه در این معماری طوری نوشته می‌شود که دیتا از بالایی‌ترین لایه (View) به سمت پایینی‌ترین لایه (Model) که معمولا دیتابیس ما هست حرکت می‌کند. در این مسیر هر لایه taskی مشخص دارد که روی این دیتا انجام دهد. برای مثال چک کردن درستی داده یا تغییر دادن شکل داده برای ذخیره در دیتابیس و ... . در این معماری مرسوم است که هر لایه توسط برنامه‌نویسان مختلف نوشته می‌شود.

در حال حاضر ساختار MVC معروف‌ترین ساختار شناخته‌شده‌ی لایه‌ای است که توسط بسیاری از frameworkها ساپورت می‌شود.

از مهم‌ترین فواید این ساختار می‌توان به Separation of concerns اشاره کرد که هر لایه باید روی نقش خودش تمرکز کند. از دیگر فواید این موضوع می‌توان maintainability بالا، تست پذیر بودن، سادگی برای تخصیص نقش به هر فرد و سادگی برای ارتقای هر لایه به صورت جدا را نام برد.

خطرات احتمالی

- کد ممکن است به یک "big ball of mud" در صورت مرتب نبودن و یا درست نبودن نقش‌ها و مسئولیت‌ها تبدیل شود.
- با توجه به آنتی-پترن "sinkhole" ممکن است کد بسیار کند شود و بخش عظیمی کار بدون توجیه برای انتقال داده بین لایه‌ها صورت گیرد.
- جدا بودن لایه‌ها از هم (با این که هدف این معماری است) ممکن است باعث شود که فهمیدن منطق کد بدون دانستن همه‌ی لایه‌ها کار بسیار سختی شود.
- ممکن است برنامه‌نویسان به اشتباه لایه‌ها را با هم ترکیب کنند و coupling بالایی به وجود آورند.

- اگر با برنامه پیش نرود ممکن است تغییرات کوچک در برنامه نیاز به deployment کامل از اول پیدا کند.

بهترین استفاده برای...

- اپلیکیشن‌های جدیدی که نیاز به پیاده‌سازی سریع دارند.
- اپلیکیشن‌های سازمانی و بیزنسی که دارای دپارتمان و پروسس سنتی IT دارند.
- تیم‌هایی که توسعه‌دهندگان کم‌تجربه دارند که بقیه‌ی معماری‌ها را در حال حاضر متوجه نمی‌شوند.
- اپلیکیشن‌هایی که نیاز به maintainability و testability بالا دارند.

معماری رویداد-محور (Event-driven)

بسیاری از برنامه‌نویسان در کار خود منتظر اند تا اتفاقی درون برنامه بیافتد. بعضی وقت‌ها دیتا باید پروسس شود و بعضی وقت‌ها نیازی به آن نیست.

معماری رویداد-محور با ساختن یک مرکز اصلی که دیتا را قبول می‌کند و آن‌ها را به ماژول‌های مختلف موکول می‌کند این مشکل را مدیریت می‌کند. این تحویل دادن دیتا را با ساخت یک event و موکول کردن آن به کدی که مربوط به آن تایپ است انجام می‌شود. در این معماری ماژول‌های جدا تنها با eventی که برای آنها مهم است سر و کار دارند و برخلاف مدل لایه‌ای، دیتا در تمامی طول برنامه حرکت نمی‌کند.

این معماری بسیار برای محیط‌های complex تطبیق‌پذیر هستند و به راحتی scale می‌شوند و هر گاه نیاز شود به سادگی یک event-type جدید برای نیاز جدیدمان تولید می‌کنیم.

خطرات احتمالی

- تست کردن در این معماری برای ماژول‌هایی که روی هم تاثیر می‌گذارند بسیار پیچیده و سخت است.
- هندل کردن errorها می‌تواند کار پیچیده‌ای شود، خصوصا وقتی که چند ماژول روی یک event کار می‌کنند.
- وقتی یک ماژول fail می‌شود، هسته‌ی مرکزی باید یک پلن جایگزین برای خود داشته باشد.
- سرعت پروسس ممکن است برای هندل کردن messageها پایین بیاید، زیرا هسته‌ی مرکزی باید یک بافر از این پیام‌ها تولید کند تا به ماژول مورد نیاز بفرستد که ممکن است زمان‌گیر شود.
- توسعه‌ی یک ساختار داده در سطح سیستم عملا بسیار سخت است، زیرا هر ماژول و event ممکن است نیازهای بسیار متفاوتی از هم را داشته باشند.
- یک سیستم انتقال باثبات درون این معماری بسیار سخت پیاده می‌شود، زیرا ماژول‌ها بسیار از هم decoupledاند.

بهترین استفاده برای...

- سیستم‌های آسنکرون با دیتافلوی آسنکرون.
- اپلیکیشن‌هایی که بلاک‌های دیتای مشخص با تعداد محدودی ماژول سر و کار دارند.
- اپلیکیشن‌هایی که در آن‌ها user-interface خیلی اهمیت دارد.

معماری میکروکرنل (Microkernel)

بسیاری از اپلیکیشن‌ها یک سری کار اصلی دارند که به صورت متداول با پترن‌های متفاوت و دیتای مختلف باید آن را انجام دهند. برای مثال یک IDE فایل را باز می‌کند، تغییر می‌دهد و کارهای بک‌گراندی که برای run این اپلیکیشن باید انجام شود را راه می‌اندازد. یک IDE تمامی این کارها را انجام می‌دهد و با زدن یک دکمه کد کامپایل شده و اجرا می‌شود. در این مورد این کارهای متوالی برای نشان دادن و ادیت کردن فایل بخشی از میکروکرنل است. کامپایلر یک بخش اضافی در کنار کار است که فیچرهای داخل میکروکرنل را ساپورت می‌کند. خیلی از وقت‌ها می‌شود یک IDE را برای زبان‌های دیگر extend کرد و با دادن یک کامپایلر به زبان مورد نظر از این IDE استفاده کرد. همچنین می‌شود که یک نفر اصلا از کامپایلر استفاده‌ای نکند و تنها برای باز کردن و تغییر دادن فایل از آن استفاده کند!

این فیچرهای اضافه که معمولا روی کار سوار می‌شوند به اسم plug-in شناخته می‌شوند. برای همین بعضی افراد به این نوع معماری plug-in architecture نیز می‌گویند. در این نوع معماری راه‌حل این است که بعضی کارهای ساده را در میکروکرنل راه‌اندازی کنیم. بعد از آن واحدهای کاری متفاوت می‌توانند plug-in مربوطه‌ی خود را روی آن سوار کنند تا کاری که می‌خواهند را (به همراه استفاده از تابع‌های میکروکرنل) انجام دهد.

خطرات احتمالی

- این که تصمیم بگیریم جه فیچری برای میکروکرنل است یک جور هنر محسوب می‌شود! این تکه باید مربوط به کاری باشد که به صورت مداوم در برنامه انجام می‌شود.
- Plug-inها باید به اندازه‌ی کافی دارای handshaking باشند تا میکروکرنل متوجه شود که یک افزونه (plug-in) روی آن نصب شده.
- وقتی تعداد خوبی از افزونه‌ها روی میکروکرنل سوار شوند، تغییر و ارتقای میکروکرنل بسیار می‌تواند سخت شود. در بعضی موارد حتی شاید مجبور شویم افزونه‌ها را نیز modify کنیم.
- انتخاب کارهای ریزدانه برای کرنل بسیار سخت و تقریبا غیرقابل تغییر بعد از دیپلوی است.

بهترین استفاده برای...

- ابزارهایی که توسط افراد متنوعی استفاده می‌شوند.
- اپلیکیشن‌هایی که بین کارهای معمولی و basic و کارهای پیشرفته‌تر خط تمایز درستی کشیده‌اند.
- اپلیکیشن‌هایی که یک سری کارهای هسته‌ای اصلی ثابت دارند و یک سری قانون dynamic که باید به صورت متداوم آپدیت شوند روی آن‌ها سوار می‌شوند.

معماری میکروسرویس (Microservices)

همانطور که می‌دانیم یک اپلیکیشن وقتی که تازه شروع به درست شدن کرده، خیلی راحت و در دسترس برای هر تغییر و ارتقایی است؛ اما به محض آن که برنامه بیش از حد بزرگ شود تلاش برای این که یکپارچه (monolithic)، غیرقابل انعطاف و ... نشود بسیار زیاد می‌شود. معماری میکروسرویس برای مدیریت کردن اینجور پروژه‌ها آمده است. در این معماری ما به جای این که یک برنامه‌ی بزرگ بسازیم، چندین برنامه‌ی کوچک می‌سازیم که تجمیع آن‌ها برنامه‌ی بزرگ‌تر درون ذهن ما را به ما بدهد و هر بار که خواستیم feature جدیدی اضافه کنیم به برنامه، با ساختن یک ریزبرنامه‌ی دیگر این کار را انجام می‌دهیم.

در این مدل انواع و اقسام کارها در برنامه‌های متفاوت پردازش می‌شوند. برای مثال در یک اپلیکیشن streaming، لیست فیلم‌های مورد علاقه، امتیازدهی به فیلم‌ها و اطلاعات حساب کاربری در سرویس‌های جدا صورت می‌گیرند. انگار که این برنامه، کالکشنی از چندین برنامه‌ی دیگر است که درنهایت یک سرویس بزرگ را به ما می‌دهد.

این روش بسیار شبیه روش‌های میکروکرنل و رویداد-محور است، با این تفاوت که تسک‌های مختلف معمولا به طرز آسانی از هم دیگر قابل تمایز اند. در بسیاری از مواقع تسک‌های مختلف می‌توانند زمان پروسس بسیار متنوعی داشته باشند و کارایی آن‌ها نیز بسیار فرق کند. ممکن است بعضی سرویس‌ها در تایم‌های مشخصی منابع بسیار زیادتری نیاز داشته باشند (مثلا دیدن فیلم در روزهای تعطیل هفته) و این برنامه‌ها باید آماده scale کردن این ریز سرویس‌ها را داشته باشند و در صورت نیاز یکی را scale up و دیگری را scale down کنند.

خطرات احتمالی

- سرویس‌ها باید به صورت بسیار بزرگی از هم مستقل باشند. در غیر اینصورت ممکن است این تداخل باعث شود که فضای ما از بالانس خارج شود.
- تمامی اپلیکیشن‌ها تسک‌هایی ندارند که بشود به راحتی به واحدهای کاری متفاوت آن‌ها را شکاند.
- پرفورمنس در این روش می‌تواند به شدت کاهش پیدا کند؛ زیرا در این روش ما هزینه‌ی communication را نیز داریم.
- تعداد بسیار زیاد میکروسرویس‌ها می‌تواند مخاطب را گیج کند، چرا که ممکن است یک بخش از صفحه بسیار دیرتر از بخش‌های دیگر بارگذاری شود.

بهترین استفاده برای...

- وبسایت‌هایی با کامپوننت‌های کوچک.
- مرکز داده‌های مشترک با مرزهای خوب تعریف شده.
- اپلیکیشن بیزنسی‌ای که باید به سرعت خیلی بالایی توسعه داده شود.
- تیم‌های ایجادی که بسیار از هم دور اند، برای مثال در شهرها و کشورهای مختلف هستند.

معماری مبتنی بر فضا (Space-based)

بسیاری از اپلیکیشن‌ها حول یک دیتابیس ساخته شده‌اند و تا زمانی که دیتابیس به خوبی پاسخ برنامه را بدهد این اپلیکیشن‌ها به خوبی کار می‌کنند. اما اگر دیتابیس به هر دلیلی (اعم از درخواست زیاد و ...) نتواند پاسخ سیستم را به حد کافی بدهد، کل برنامه می‌تواند fail شود.

معماری Space-based با جدا سازی پروسس و ذخیره‌سازی بین سرورهای مختلف، طراحی شده تا زیر فشار و لود زیاد ما در برنامه functional collapse نداشته باشیم. دیتا در این جا بین nodeهای مختلف پخش می‌شود. این روش نودها را پیکربندی می کند و طبق گفته‌ی نویسنده‌ی کتاب معماری مبتنی بر فضا چیزهایی که خیلی غیرقابل پیش‌بینی و خطرناک اند را از دیتابیس حذف می‌کند. در این معماری ما اینجور اطلاعات را در RAM ذخیره می‌کنیم که باعث می‌شود انجام کارها بسیار سریع‌تر شوند. اما اینجور ذخیره کردن اطلاعات می‌تواند آنالیز کردن را بسیار پیچیده کند. در این معماری بعضی حساب‌کردن‌ها با

دیتا ممکن به دیتاهای مختلفی نیاز داشته باشد که دسترسی به همه‌ی آن‌ها سخت می‌شود و بعضا حساب‌کردن باید به چند کار کوچک بشکند.

خطرات احتمالی

- ساپورت انتقال اطلاعات با دیتابیس درون RAM بسیار سخت‌تر است.
- تولیدکردن لود لازم برای تست‌کردن سیستم می‌تواند چالش‌برانگیز باشد. اما هر نود جدا می‌تواند به صورت مستقل تست شود.
- توسعه‌ی مهارت لازم برای cache کردن دیتا و سرعت بخشیدن به کار، کمی سخت است و دانش خوبی را می‌طلبد.

بهترین استفاده برای...

- مقادیر زیاد دیتا مثل user logها.
- مقادیر کم دیتا که ممکن است به صورت دوره‌ای از دست بروند و تبعات منفی بدی ندارند (مثلا تراکنش‌های بانکی به هیچ عنوان نباید ازین مدل استفاده کنند)
- شبکه‌های مجازی

انتخاب اولیه‌ی ما برای معماری

با توجه به برنامه‌ی محول شده به ما باید موارد زیر را در نظر بگیریم:

- کدام یک از ریسک‌های معماری‌ها را نمی‌توانیم بپذیریم؟
  - با توجه به این که ما نیاز به نگه‌داری دیتا هستیم (هم باید دیتای کاربران را نگه‌داریم و هم باید دیتای تسک‌هایی که کاربران درست می‌کنند را نگه داریم) بنابرین هیچ‌جوره نمی‌توانیم خطرات معماری مبتنی بر فضا را بپذیریم.
  - همچنین از آنجایی که قرار نیست افزونه‌ای روی کار روتین ما اضافه شود، معماری میکروکرنل نیز برای پروژه‌ی ما بیهوده تلقی می‌شود. در این پروژه ما کار basicی را نمی‌توانیم متصور شویم که روی آن بقیه تسک‌های dynamic را سوار کنیم.
  - از آنجایی که ما eventهای متعدد زیادی نیز نداریم، به نظر میاید ریسک‌های معماری رویداد-محور را نیز بهتر باشد نپذیریم. این معماری خصوصا به دانش نسبتا خوبی نیاز دارد که شاید تیم توسعه‌ی فعلی ما این دانش را به حد کافی برای ماژول‌های مختلف درست کردن ندارد.

- کدام معماری می‌تواند برای ما بهتر باشد؟
  - از بین دو معماری باقی مانده، هر دو برای تیم‌های تازه‌کار می تواند خوب و مفید باشد که این برای ما یک نکته‌ی مثبت محسوب می‌شود.
  - با توجه به این که ما در کل می‌توانیم دو بخش user management و task management را برای پروژه در نظر بگیریم، شاید معماری میکروسرویس برای ما خوب باشد.

- از طرف دیگر تیم ما دارای دو فرد با experience در قسمت back و دو فرد با experience در قسمت front است. ازین رو تقسیم شدن تیم در معماری لایه‌ای بسیار ملموس‌تر و راحت‌تر می‌تواند صورت بپذیرد.
- با توجه به این که ما در قسمت back از فریمورک جنگو استفاده می‌کنیم، معماری لایه‌ای که درون آن به صورت پیش‌فرض تعبیه شده است.
- تست کردن در معماری لایه‌ای تا حدی پیچیده‌تر است و باید خیلی وقت‌ها با تغییراتی که در سیستم می‌دهیم کل سیستم را تست کنیم. این ریسک در معماری میکروسرویس کمتر است و تقریبا هر سرویس را جداگانه می‌شود بررسی کرد و بعدا تعاملش به صورت جداگانه نیز با سرویس‌های دیگر بررسی شود.

بنابرین در تیم، ما با دانستن مخاطرات (ریسک‌های) مربوط به پیچیده شدن وحشتناک کد، بر این مورد تصمیم گرفتیم که معماری را لایه‌ای پیش ببریم و اگر موفق بود و ریسک‌هایش را توانستیم manage کنیم به همین صورت ادامه دهیم؛ در غیر این صورت معماری خود را به میکروسرویس تغییر خواهیم داد و آن را بررسی خواهیم کرد تا به proven architecture مورد نظر خود برسیم.

- ❖ برای آن که ریسک رد نشدن از ددلاین را نیز کم کنیم در فاز اول تصمیم گرفتیم که ریسک را با انجام بخش user management بسنجیم و درصورت موفق بودن در فازهای بعدی عملیات‌های دیگر مورد نیاز را نیز پیاده کنیم.

معماری نهایی (Proven Architecture)

بعد از پیشرفت در فاز اول ما تصمیم گرفتیم با معماری لایه‌ای کد خود را جلو ببریم.

- مهم‌ترین دلیل built-in بودن این معماری در فریم‌وورک جنگو که ما با آن کار می‌کنیم می‌تواند باشد. این انتخاب به ما این اجازه را می‌دهد که از پتانسیل کامل تکنولوژی خودمان استفاده کنیم و پروژه را به بهترین نحو جلو ببریم.
- پروژه‌ی ما، پروژه‌ی عجیب غریب جدیدی نیست که بخواهیم خیلی به پروژه‌های ریزتر بشکانیم و هر کدام را جداگانه بزنیم. توجیح استفاده از معماری میکروسرویس با وجود تنها 2 سرویس (مدیریت کاربران و مدیریت پروژه) خیلی منطقی نظر نمی‌کرد، بنابرین ما ترجیح دادیم ریسک‌های این معماری را نپذیریم.
- با توجه به این که با معماری لایه‌ای کد ممکن است به درهم‌تنیدگی بسیار برسد، ما در فرانت بین دو چارچوب React و Angular از Angular استفاده می‌کنیم که باعث می‌شود بتوانیم به صورت component اجزای خود را جدا کنیم و اجزای درشت‌دانه و ریزدانه داشته باشیم که سلسه‌مراتب آن قابل تشخیص باشد و در صورت بروز مشکل بتوانیم با trace کردن این componentها راحت‌تر منبع مشکل را پیدا کنیم.
- با توجه به فرصت کمی که برای پروژه داریم ما ترجیح دادیم سراغ تکنولوژی جدید تا جای ممکنه کم‌تر برویم و با تکنولوژی‌هایی که با آن‌ها آشنا هستیم کار کنیم و در صورت نیاز دانش خود را در این تکنولوژی‌ها ارتقا دهیم و با معماری انتخابی هماهنگ کنیم. اینگونه ریسک نرسیدن به ددلاین را نیز کمتر کرده‌ایم.
- با توجه به این که تیم ما توسعه‌دهنده‌های حوزه‌های مختلف دارد (فرانت و بک جدا) بنابرین لایه‌ای پیش رفتن مزیت دارد و باعث می‌شود که هر کس با هماهنگی APIها روی نقطه‌ی قوت خود کار کند و در صورت لزوم نیز به قدری دانش برای وقتی که الزام به swarming شود را نیز داریم.

- همچنین ریسک بی‌برنامگی که ممکن است منجر به خرابی شود را نیز با جلو بردن کارها در ابزار مدیریت پروژه‌ی Jira تا جای ممکنه مدیریت می‌کنیم.

بنابرین پس از کار کردن در فاز اول پروژه ما با پذیرفتن ریسک‌های **معماری لایه‌ای** و سعی در حداقل کردن آن‌ها، این معماری را انتخاب کردیم و در ادامه نیز با همین معماری پروژه‌ی خود را جلو خواهیم برد.

منابع

- https://techbeacon.com/app-dev-testing/top-5-software-architecture-patterns-how-make-right-choice
- https://www.allerin.com/blog/software-product-development