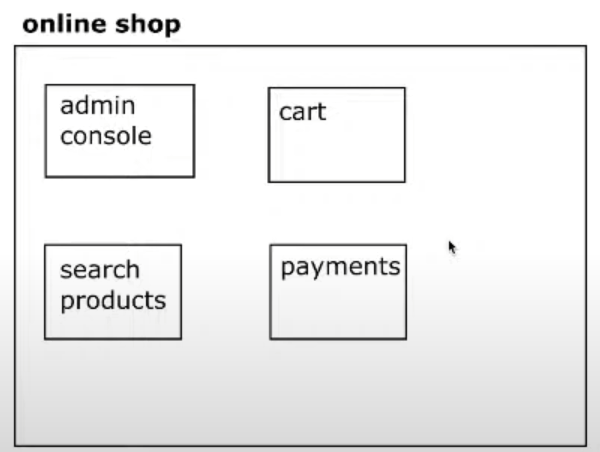
**Monolith**

Считается самым старшим архитектурным стилем

В этом стиле есть всего 1 процесс, который осуществяет множество обязанностей (функциональностей)



Консоль админа, корзина пользователя, оплата – все эти функциональности находятся вместе

Если взять в пример Java и фреймворк Sprint, то весь проект – это архив с .jar файлом, и мы можем его запустить и это и будет наш монолит

В таком случае мы можем либо запустить все функциональности, либо никакую, так как они запускаются вместе

Монолитная архитектура считается legacy, так как раньше разработчики не знали принципов разработки (SOLID, DRY, …) и теперь бОльшая часть старых приложений, которая является монолитной – написана по современным меркам коряво. Так же раньше не было других типов архитектур SOA, Microservices, так как коммуникация между сервисами в данных архитектурах осуществляется посредством интернета, а раньше он был медленный. Так же не было контейнеризации (Docker), который бы позволял быстро разворачивать сервисы

Context

You are developing a server-side enterprise application. It must support a variety of different clients including desktop browsers, mobile browsers and native mobile applications. The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker. The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response. There are logical components corresponding to different functional areas of the application.

Problem

What’s the application’s deployment architecture?

Forces

* There is a team of developers working on the application
* New team members must quickly become productive
* The application must be easy to understand and modify
* You want to practice continuous deployment of the application
* You must run multiple instances of the application on multiple machines in order to satisfy scalability and availability requirements
* You want to take advantage of emerging technologies (frameworks, programming languages, etc)

Solution

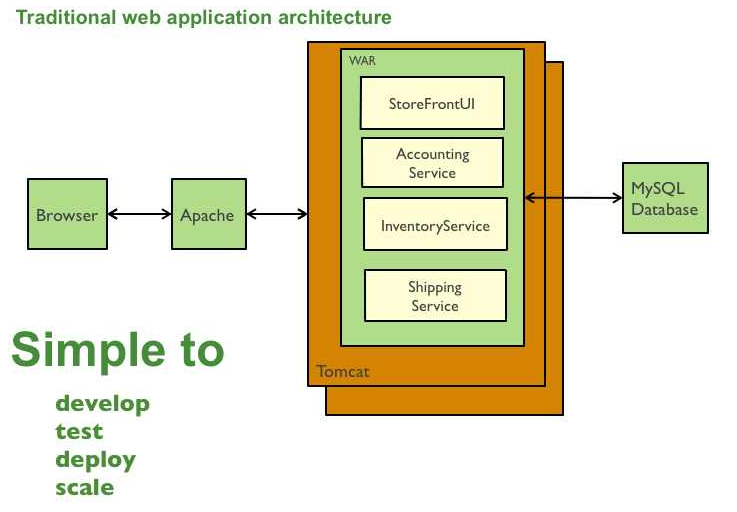
Build an application with a monolithic architecture. For example:

* a single Java WAR file.
* a single directory hierarchy of Rails or NodeJS code

Example

Let’s imagine that you are building an e-commerce application that takes orders from customers, verifies inventory and available credit, and ships them. The application consists of several components including the StoreFrontUI, which implements the user interface, along with some backend services for checking credit, maintaining inventory and shipping orders.

The application is deployed as a single monolithic application. For example, a Java web application consists of a single WAR file that runs on a web container such as Tomcat. A Rails application consists of a single directory hierarchy deployed using either, for example, Phusion Passenger on Apache/Nginx or JRuby on Tomcat. You can run multiple instances of the application behind a load balancer in order to scale and improve availability.



Resulting context

This solution has a number of benefits:

* **Simple to develop** - the goal of current development tools and IDEs is to support the development of monolithic applications
* **Simplicity in deployment** (нужно конфигурировать только 1 систему, не нужно настраивать коммуникацию между сервисами так как у нас по сути только 1 сервис, не нужно разворачивать докер контейнеры для каждого сервиса и оркестрировать их)
* **Simple to scale** - you can scale the application by running multiple copies of the application behind a load balancer
* **Communication using methods**. Что бы какой то сервис вызвал другой сервис (например при покупке товара смотрится что есть в корзине), то на бэкенде происходит просто вызов какого то метода, по типу buy\_products(), в случае же с микросервисами, коммуникация должна идти по сети через API, что не надежно

However, once the application becomes large and the team grows in size, this approach has a number of drawbacks that become increasingly significant:

* **The way app is written** - The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Also, because there are not hard module boundaries, modularity breaks down over time. Moreover, because it can be difficult to understand how to correctly implement a change the quality of the code declines over time. It’s a downwards spiral. Чем больше становится приложение, тем дольше оно собирается и стартует, а так же становится сложнее его разрабатывать, так как нужно держать в голове все зависимости со всеми модулями, тогда как в SOA можно думать только о зависимостях в конкретном микросервисе (принцип SOLID)
* **Overloaded IDE** - the larger the code base the slower the IDE and the less productive developers are.
* **Overloaded web container** - the larger the application the longer it takes to start up. This had have a huge impact on developer productivity because of time wasted waiting for the container to start. It also impacts deployment too.
* **Continuous deployment is difficult** - a large monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems. There is also the chance that components that haven’t been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers, since they usually need to iterative rapidly and redeploy frequently.
* **Scaling the application can be difficult** - a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application. Some clouds can even adjust the number of instances dynamically based on load. But on the other hand, this architecture can’t scale with an increasing data volume. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently. В случае с SOA, мы можем масштабировать только конкретный микросервис, который больше всего нагружается. В случае же с монолитом, мы должны замаштабировать все сервисы сразу
* **Obstacle to scaling development** - A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size its useful to divide up the engineering organization into teams that focus on specific functional areas. For example, we might want to have the UI team, accounting team, inventory team, etc. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.
* **Requires a long-term commitment to a technology stack** - a monolithic architecture forces you to be married to the technology stack (and in some cases, to a particular version of that technology) you chose at the start of development . With a monolithic application, can be difficult to incrementally adopt a newer technology. For example, let’s imagine that you chose the JVM. You have some language choices since as well as Java you can use other JVM languages that inter-operate nicely with Java such as Groovy and Scala. But components written in non-JVM languages do not have a place within your monolithic architecture. Also, if your application uses a platform framework that subsequently becomes obsolete then it can be challenging to incrementally migrate the application to a newer and better framework. It’s possible that in order to adopt a newer platform framework you have to rewrite the entire application, which is a risky undertaking.
* **Availability**. В случае с SOA, если упал какой то микросервис, то другие продолжают работать, в случае же монолита, если что то упадет, то упадет сразу все
* **Does it have a state?** Как вправлять statefull состоянием