DAY 1

הפתרון האופטימלי הוא ליצור כמה מיקרוסרוויסים בשביל שרתי הbackend ולהשתמש בLoad Balancer (LB) על מנת ליצור שוויון בחלוקת העומס. יצירת כמה שרתי backend תעזור בהפחתת עומס המשתמשים על שרת אחד, ובכך תעזור בהימנעות מקריסת האתר עקב עומס. כאשר יש כמה שרתי backend, גם אם אחד מהשרתים קורס, הוא לא יקריס את כל האתר, מכיוון שה-LB יחוון את המשתמשים לשרת backend אחר פעיל.

ישנם כמה סוגים לLoad Balancer: Round Robin, Weighted Round Robin, IP Hash, Least Connection Algorithm, Least Response Time Algorithm וכו'. לכל אחד מהם יתרונות וחסרונות. במערכת גדולה כמו אתר e-Commerce שאנחנו עתידים לבנות, הייתי מעדיפה להשתמש בשיטה של Least Connection Algorithm. בשיטה זו הLB בודק באיזה שרת יש פחות active connections ואליו שולח את המשתמש. כך יש התפלגות שווה של משתמשים ועומס בין השרתים, ומונע עומס על שרת ספציפי אחד. ישנם גם חסרונות לשיטה זו, כמו למשל שהיא לא מתחשבת ביכולות השרת והביצועים שלו, אך בביצוע אידיאלי של האתר אני שואפת לכך שמשאבי השרתים יהיו זהים, כך שזה לא ישנה. כמו כן, הבקשות באתר לא אמורות להיות כבדות במיוחד, ולכן לא אמורות להיות בקשות שיצרו עומס מיוחד על השרתים.

מכיוון שאין הרבה זמן, וגם העומס על האתר שאבנה לא אמור להיות כבד, לא אעשה זאת באתר, ויהיה רק שרת backend אחד.

Horizontal Scaling – יצירת עוד nodes כדי לחלק את העומס (למשל, יצירת עוד שרתי backend)

Vertical Scaling – הוספת משאבים לשרת אחד. למשל, להגדיל את כוח ההתמודדות של השרת עם עומס. יותר קל לעשות אבל פחות חסכוני. בנוסף יש סיכון של Single Point Of Failure כי יש רק שרת אחד עם כוח גדול, ואם הוא נופל כל האתר קורס.

אני אבחר להשתמש בnode.js על מנת לבנות את שרת הbackend שלי כי יש לו יתרונות שיעזרו לי לעשות את האתר יותר סקלבילי כמו: התמודדות א-סינכרונית עם בקשות כך שניתן לקבל הרבה בקשות בו-זמנית ללא בעיה.

על מנת לשפר את זמן התגובה והטעינה:

האידיאל – להשתמש בcaching וCDN

Caching – מקום אחסון זמני לתוצאות של תגובות יקרות או של מידע שניגשים אליו בתדירות גבוהה, כדי שלבקשות בשבילן ייקח הרבה פחות זמן להחזיר תשובה ולהגיב. כאשר אנו מקבלים בקשה, שרת האינטרנט בודק אם המידע נמצא בcache, במידה וכן, הוא מחזיר את המידע משם, ובמידה ולא, הוא שולח בקשה לDB, שומר את התשובה בcache, ומחזיר את המידע ללקוח.

CDN – רשת שרתים מרוחקים שמשומשים על מנת לאחסן מידע סטטי, כמו תמונות, סרטונים, קבצי CSS, קבצי JS ועוד. כאשר הלקוח מבקש מידע סטטי, השרת האינטרנטי בודק אם הוא קיים בCDN, במידה וכן, שולח ללקוח, ובמידה ולא, מבקש מהשרת, מאחסן את התשובה בCDN, ומחזיר ללקוח.

כדי להימנע מSPOF בשכבת הDB, הפתרון האידיאלי יהיה להשתמש ב- Database Sharding – פיצול הDB לכמה DB שונים. ניתן לעשות זאת בשתי דרכים:

Data Replication - יהיה master DB ועותקים של המאסטר. במאסטר יהיה ניתן רק לכתוב, ואת העותקים שלו יהיה ניתן רק לקרוא. יתרונות השיטה: המודל מאפשר ליותר שאילתות להתבצע בו-זמנית ואם הDB נהרס פיזית, המידע נשמר, מכיוון שיש לו עותקים במקומות אחרים.

פיצול המידע לכמה DB – ניתן לעשות sharding למידע כך שבכל DB תהיה אותה טבלה, אך המידע בה יהיה שונה. Sharding עובד לפי hash מסוים שלפיו ישייך את המידע הנכנס לDB תואם. למשל, יצירת 4 DB, כך שהhash יהיה: id%4, ולפי זה ישויך למספר הDB התואם. כך, המידע יתפצל בין כמה DB ולא יהיה עומס על אחד. לשיטה זו יש כמה בעיות: 1. יכול להיות שלא תהיה חלוקה שווה של המידע בין הDB, ואחד הDB יגיע לshard exhaustion, ויצטרכו להזיז מידע ולשנות את המפתח של הhash. 2. בעיית סלבריטי – יכול להיות שבDB אחד יהיו כמה רשומות שהרבה פעמים יקראו, והDB יהיה מוצף בקריאות.

מאחר והמערכת שלי לא אמורה להיות גדולה במיוחד, אני מעדיפה לעשות sharding בצורה של data replication, למקרה ויקרה משהו לאחד הDB שלי. העומס על הDB לא אמור להיות גדול במיוחד, ולכן אין צורך בפיצול המידע בין DB שונים.

כל הדברים שכתבתי מקודם, הם דרכים להתמודדות עם שגיאות וקריסות. למשל, הרצת שרת backend במקביל, כך שאם אחד קורס יהיה שרת חלופי, העתקת המידע בDB לDB חלופי במקרה וקורה משהו לאחד מהם וכו'. בנוסף, יש אפשרות לגנרטור במקרה ותהיה הפסקת חשמל, כדי למנוע מהשרתים לקרוס.

נוסף על כך, האידיאל הוא לעשות שרתים שונים לכל service כדי שאם אחד מהם יקרוס, שאר השרתים עדיין יוכלו לתפקד (למשל: אם השרת של הקנייה נופל, שאר האתר עדיין יוכל לרוץ).

על מנת למנוע בעיות עתידיות, רצוי לעשות לוגים ובדיקות עוד לפני העלאת האתר. בעזרת הלוגים, יהיה ניתן לעקוב אחר הבקשות ולוודא שהכול מתנהל כראוי, ובמידה ולא, להבין איפה זה נפל. כמו כן, בדיקות כמו Unit testing יכולות למצוא תקלות עתידיות של מקרי קצה שלא שמנו לב אליהם במהלך הפיתוח.