# Scalability and Security

|  |  |  |
| --- | --- | --- |
| Topic | Command | Info |
| Scaling methods |  | **Vertical**(bigger server) and **Horizontal**(more servers) scaling + load balancer |
| Session-aware scaling |  | Sticky sessions (remembers the server I had last time and sends me there again).  Session in DB. (save session info in db. Which is accessible from all servers)  Client-side sessions (store info in cookies and send them to session) -> Problems: fakeabale cookies, a lot of info being sent back and forth |
| Autoscaling |  | How many servers do we need? Use autoscaling and start with 2 servers. Choose min and max nr. of servers. If there is enough traffic on the website, scale up etc. |
| Heartbeat Server |  | Check if servers are still functioning or not. Send a heartbeat signal to all servers check if they work. |
| Database Partiioning |  | Scalability is also important for databases because even if we have more servers they all access the same database. One approach is database **portioning (vertical)** where we split the data in different tables. (i.e. split 1 table in 2 tables: 1 only airports, 1 for mapping 🡪 smaller tables)  Another approach is the **horizontal partitioning** (split one big table into more smaller table which all store the same data – same columns – but with a different context, i.e. flights splits to flights\_international and flights\_domestic)  Lasting problem: this is still a single point of failure because if the db falls the system goes down |
| Database Replication |  | Used to fix the database portioning problem. Models:  **Single-Primary Replication**: only one DB has read/write privileges while the rest can only read. If the main DB is changed it sends this information to all the other ones. Problem is still kind of the same because the single point of failure here is the write point  **Multi-Primary Replication:** All the replicated DB have read/write privileges but they must all update themselves. Potential problems: conflict problems: update conflicts: what if 2 people edit the same row at the same time? Or uniqueness conflicts: what if top people want to insert data at the same time |
| Caching | Cache-Control: max-age: x  ETag: “someTag” | What if one of the main request is sent by a number of users which always has the same query, i.e. main page of New York Times where the main page loads a lot of articles. For this example we can store and save the SQL request such that accessing it becomes much faster.  **Client-Side Caching:** Store the request locally on the user’s computer. Cache-control can be used for that where the current version is cached for x seconds and if we request it within this time we show the cached version, else request the new version from the server. We can also use ETag to track the version, so if the ETag Is different load new version always.  **Server-Side Caching:** Per-View Cache: cache specific views  Template Fragment Caching: Cache a part of the template  Low-Level Cache API: save results inside of a separate Cache |
| Security | HTTPS. Use public key to create ciphertext and private to decrypt it | **Public key encryption**: we have 2 keys, public and private, where public is used to encrypt information and the private one is used to decrypt information. Public key is sent to user to encrypt the data and private key always stays with you and Is used to decrypt the message. |
| DB Securiy | User data: save username as username and password hashed | The hashed function is a one-way function, so in the table we don’t know what the password actually is. The password can be changed but not read server side |
| SQL injection security |  | User and PW example where user is hacker “-- ; this means ignore rest in SQL. Django however escapes those characters to make sure that this is not possible |
| API Request Security |  | **Rate Limiting**: limit the amounts of api requests per mine to not crash the server  **Route Authentication**: only allow certain can access the API by using an API Key (essentially a pw) that needs to be passed to the API. Make sure to not put the API Key in the git repo. The API key can be set as an environment variable so that the key is drawn from there. |
| JS Security |  | **Cross-Site Scripting**: Someone else can write their own JS code and run it on your website. In general on your application you only want your JS code to run if you yourself have written it. |
| DB Manipulation | Only use POST requests to manipulate something in the data | Having GET requests that can manipulate data in the DB is a bad practice due to cross-site forgery. Best practice here is to always have a put request as it can be verified client-side in combination with CSRF tokens because if someone else wants to make a forged request they will not know what the CSRF token is (as it changes every session) |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |