**Ingeniería Web**

Regularizar: mas de 60 en ambas partes

Promoción: mas de 80 en ambas

Horario: 18

https://bitbucket.org/ucsedarseia/seia-catedra/wiki/Clases2017

Clase 1

**HTTP**

Servidor – Cliente por separado (desconectadas).

El servidor no pide cosas al cliente directamente. El servidor envía responses cada vez que el cliente realiza una petición (request). Para comunicarse se utiliza el protocolo HTTP. Las peticiones indican que documento quieren ver y el servidor responde.

Protocolo http: define como se comunican. La conexión se abre y se cierra constantemente.

**Petición:** empieza con un título que indica que es lo que se quiere hacer (verbos get, post) y sobre qué documento o dirección.

Get/post | Versión de protocolo HTTP |url después del dominio

Headers (data extra, por ejemplo no cache, user agent, resolución de pantalla, referred, formatos que acepto). Le sirve al servidor para devolver una mejor respuesta.

Body (se coloca data solo si se quiere subir algo al servidor, sino vacío).

**Response:**

Versión | Código de error (200 ok, 300 la página está en otro lado, 404 not found, 500 mal)

Headers

Body (contenido que se debe mostrar al cliente, HTML para mostrar documentos, imágenes, videos, formatos para aplicaciones específicas JSON). El cliente debe poder interpretar la response.

**Problemas:**

* Falta de estado: después de cada función, la petición se cierra. Después de esto el servidor no sabe que es el mismo cliente el que vuelve a pedir. Posibles soluciones: enviar la dirección IP (ya no se usa, no sirve), que el servidor devuelva links con identificador de usuario (se usa muy poco, no sirve). Mejor solución: guardar la sesión y algún dato de cada petición. El servidor devuelve una cookie con la sesión del usuario para poder identificarlo en futuras peticiones. Luego cada vez que el cliente realiza una petición, envía la misma cookie. Las cookies ocupan muy poco espacio y no se le permite al usuario realizar grandes modificaciones.
* Algunos sitios como Google y Facebook recolectan cookies de usuarios que visitan otras páginas. Las cookies violan la privacidad, y a veces se utilizan para publicidad.
* Para que otra persona no puede utilizar nuestra cookie se utiliza https (conexión encriptada que solo el cliente y servidor pueden entender. Insume tiempo y recursos). En las responses también se utilizan certificados para validar que sea el dueño del dominio el que responde. El certificado incluye una clave privada y una pública. La privada se utiliza para cifrar los datos, con esta clave se firman las cosas.

Las cookies sirven para guardar el ID del usuario pero también se puede utilizar para otras cosas. Las cookies se pueden utilizar para dominios o subdominios y además se pueden usar por ventana o en general.

Al ingresar a una página se realizan varias peticiones para cada elemento, pero algunos sitios modernos mantienen abierta la conexión para no hacer tantas peticiones.

Ejemplo petición:

GET / HTTP/ 1.1

Host: [www.loiprocesos.com](http://www.loiprocesos.com)

Connection: keep-alive

Cache-Control: no-cache

Ejemplo response:

HTTP/1.1 200 OK

Connection: keep-alive

Server:

Date:

Para no escribir el body se ponen dos enter (peticiones).

Para diseñar:

.NET se utiliza MVC.

En Python se usa Django. Se puede usar la consola de Linux o PyCharm.

**Introducción a Django**

* Framework de desarrollo Web: herramientas ya diseñadas para desarrollo web en Python.
* Desarrollado en Python.
* Software Libre
* Comunidad: muchas ayudas, guías.
* “El framework web para perfeccionistas con fechas de entrega”: para hacer las cosas rápidas y bien hechas.

Nuestros trabajos van a ser un proyecto (nombre del sitio) más aplicaciones independientes (grupos de funcionalidades del sitio).

Las **aplicaciones** contienen:

* Modelos: cosas que se conectan a las bases de datos.
* Vistas: lógica del sitio para responder al usuario.
* Controlador: definen cuando se ejecutan las vistas (URL y función).

**Herramientas**

ORM: automatiza el intercambio entre la aplicación y la base de datos usando SQL. ORM sirve para todas las bases de datos así que se puede mantener el mismo modelo.

Plantillas: sirven para facilitar la creación del HTML colocando los elementos donde el desarrollador quiere que se vean.

Manejo de sesiones: el manejo de cookies se realiza automáticamente.

Manejo de cache: herramienta que decide si cargar la página desde caché o volver a generar la página.

Gestión de configuración: base de datos, tablas, ROM.

Internacionalización: que se vea bien en cualquier parte del mundo (idiomas, símbolos).

Formularios: permite leer más fácil lo que el cliente ingresa en los textbox.

Testing.

**Utilidades de desarrollo**

* Servidor de desarrollo
* Consola de Django
* Modo Debug

**Djando Admin**

Genera ABM de usuario automáticamente. Sirve para desarrollo y producción.

Clase 2

**Justificar el proyecto**

No debería haber otra cosa similar. Si no aporta nada, no vale la pena hacer el proyecto.

Analizar la competencia en internet y fuera, con qué sitios se compara y por qué es distinto. La justificación puede ser técnica, social o comercial.

Qué esperamos del sitio.

**Identificar los usuarios**

Quiénes son, qué hacen, cuáles son los usuarios generales y particulares (edad, conocimiento).

En el parcial se pide especificar un usuario particular: nombre, edad, conocimiento, desde donde utiliza el sitio.

**Escenarios de uso**

Explicar las tareas importantes que el usuario puede realizar en el sitio.

Ejemplo sitio sobre espadas:

No existe ninguna comunidad online. Todavía no hay ningún sitio

Para atraer más gente.

Podría mejor la comunicación entre los usuarios pero hay que justificar bien porque puede que se comuniquen por wapp.

Indicar las desventajas 🡪 Costos de producción del sitio. Mantenimiento. Si hay poca gente interesada, no convendría hacer el sitio.

Indicar otras alternativas que existen 🡪 reuniones, revistas (si son pocas, indicar cuales), videos en youtube, foros.

Usuarios generales 🡪 gente rara que les gustan las espadas, fabricantes de espadas, interesados en aprender, interesados en enseñar, curiosos.

Usuarios concretos (por lo menos 2) 🡪 juan, se dedica a hacer espadas, entra para buscar clientes y ofrecer sus productos, entra desde una pc de escritorio la mayor parte del tiempo, espera conseguir clientes.

Escenarios de uso (por lo menos 3): Registrarse (acceder a la página, hacer clic en un botón, ingresar usuario, mail y contraseña, clic para guardar los datos, enviar mail de confirmación, confirmar mail, mostrar página de que se registró correctamente.)

**DNS, registro del nombre del proyecto**

**Derechos de copia del SW**

Definir un contrato para indicar quién es el dueño del sitio, del contenido del sitio y de la idea del proyecto.

Buscar una licencia para usar.

Clase 3

**Estructura gruesa del sistema**

Usuarios:

* Información que los identifique. Que tenga algo que los haga “responsables” de lo que hagan.
* No exigir al usuario ingresar datos no relevantes para el sitio.
* Metadata sobre cómo llegó al sitio y sobre el uso que le da.

BD usuarios: no se guarda la info directa, no se encripta. Usar **Hash**.

Contenido:

* Una tabla o muchas.
* Versionado del contenido.
* Puede interesar guardar información sobre estados, modificaciones, historia de denuncias, etc.
* Dueño, permisos, visibilidad, etc.
* Fecha, vencimiento, etc.
* Metadata, resumen, descripción, clasificaciones, tipo, etc.

Mapeo usuarios-contenido:

* Algunas cosas de lo anterior.
* Gusta o no gusta.
* Quién comentó, respondió, posteó, leyó.
* Interés en ser notificado de respuestas o comentarios.

Mapeo usuarios-usuarios:

* Agrupaciones (geografía, intereses, amistad, departamentos, etc.)
* Filtros y relaciones individuales.

**Inyecciones de SQL**

Un usuario tipea SQL cuando ingresa información. Puede acceder a la BD si las sentencias se escriben como string en el código.

Solución: usar ORM.

Fat vs Skinny (modelo de datos de usuario)

Modelo Fat no ocupa más espacio, las db son inteligentes. Es feo cuando se suman más y más campos opcionales. Siempre modificando la tabla, las consultas, etc.

Modelo Skinny se lleva mejor con eso, no hay que mantenerlo. Se usa una tabla en la que se agrega el usuario, el campo y el valor. No se agregan más columnas.

Es más complicado de programar y hacer consultas. Se usa cuando los campos de las tablas varían mucho en el tiempo.

**Bloque usuario**

Aceptación:

* Definir flujo de registración y requerimientos. Formularios cortos.
* Guardar el estado en un único campo de estado, no muchas columnas de ya hizo tal paso.
* No limitar totalmente el uso del sitio cuando no está registrado.

Flujos de páginas:

* Utilidades: diagramar, entender pasos.
* Pocos pasos, evitar complicar al usuario.
* Pensar todos los caminos (qué pasa si se sale antes de aceptar).

Flujos de estados: [se suele pedir en los parciales/finales.]

Diagrama transición de estados.

Usar un código para manejar los estados. No agregar columnas.

Se modelan por un lado los estados de un objeto y por otro las páginas que ve el usuario con las acciones que realiza.

El diagrama de páginas sirve para entender si los pasos para realizar algo son muy complicados.

**Bloque contenido**

Puede tener sentido crear una sola tabla que tenga todo el contenido porque las cosas que se publican suelen tener los mismos campos. El problema es que, si el sitio es grande, la tabla es enorme.

Suele ser necesario tener una tabla historial de los cambios que se realizan. (fecha y autor)

Si el contenido del sitio necesita versionado:

* Opción 1: tener las versiones dentro de la tabla de contenido.
* Opción 2: tener una tabla separada con todas las versiones.

**Versionado de código y servidores**

Puede ser necesario tener diferentes versiones funcionando.

Se tienen 3 servidores de aplicación y de contenido: desarrollo, test y producción.

Test tiene que ser lo más parecido a producción. En test se suele usar un backup de la bd de producción. En dev se tiene una bd de prueba.

Se usan branches para tener versiones paralelas de código.

**Get y post**

La única forma de que no se vean los datos es con https. Encriptar la request.

Request Formulario Get

<form action=”/login” method=”get”>

Request Head: Misitio.com/login?usu=fisa&pwd=paparulo

Request Formulario Post

<form action=”/login” method=”post”>

Request head: Misitio.com/login

Body: usu=fisa&pwd=paparulo

<form action=”/login” method=”post”>

<label for="usu">Usuario:</label> /\* una opción \*/

<input type="text" name="usu" value="fisa"/> /\*value no es necesario\*/

<input type="submit" value=”ok”></input>

</form>

Clase 4

**Modularidad de SW**

Los módulos deberían tener:

* Definición de BD.
* Lógica de módulos compartidos.
* Set de páginas que el módulo muestra.
* Documentación.

Ejemplos de módulos: (apps)

* Registro y manejo de usuarios.
* Manejo de contenido.
* Manejo de foros.
* Interacción con redes sociales.

No siempre es simple diferenciar los distintos módulos, pero es necesario dividir.

Facilita encontrar las cosas si hay errores, por ejemplo.

Dentro del módulo también hay que seguir cierta estructura (carpetas, nomenclatura).

Documentación:

* Explicación general.
* Descripciones de métodos, parámetros.
* Resumen de las cosas públicas que tenga el módulo.
* Configuraciones.

Problema:

* Olvidarse de actualizar la documentación.
* Tiene que ser entendible.

Lógica compartida

No se suele usar meter código dentro de la BD. Queda descoenctado del resto del código.

Cuando los módulos son muy grandes, se ponen en servidores separados.

Aislar a los diseñadores

* Separar html de código en directorios distintos para que los diseñadores puedan tocar html. (usar esto)
* Poner los templates en una BD. Ventaja: se ven rápidos los cambios. Desventajas: fácil de romper. Muchas llamadas a la BD.

Parámetros de configuración

* No poner en la BD.

**Foros de discusión**

Un foro puede ser una comunidad. 6 cosas para hacer para que la comunidad dure. Capítulo Planing. **PARCIAL**

* Detectar usuarios molestos.
* Delegar moderación.
* Contenido atractivo.
* Navegación y búsqueda. Cosas públicas contenido fácil de buscar.
* Extensibilidad del SW. (ejemplo face: conectarse con face desde otros lados)
* Medios de colaboración. Los usuarios tienen que publicar.

No dividir en muchas secciones ni pocas. Es complicado encontrar lo que uno busca. Si son muchos puede que muchas queden sin uso y si son pocas, se mezclan los temas.

Hacer árboles de categorías.

También es bueno hacer un mixto: en la home se ven publicaciones mezcladas pero también hay secciones.

Tener en cuenta que todo tiene que ser fácil de usar para el usuario. Testear la usabilidad:

* Guardar estadísticas.
* Poner a un usuario real.

Foros de consultas

Es útil ofrecer la posibilidad de marcar la respuesta correcta (calificación). Ayuda a que la gente no haga las mismas preguntas y además la gente se queda porque encuentra rápido de la respuesta.

Es necesario algún tipo de incentivo para que los expertos sigan participando.

* Se lo convierte en moderador o se le da algún tipo de autoridad dentro del sitio.
* Hacer categorías de usuarios, reputación, acumulación de votos, etc. Se puede agregar alguna utilidad real.

**Ajax**

Permite hacer request al servidor sin que la página se recargue.

Opciones:

* Pedir el html a remplazar.
* Pedir solo los datos. (se usa hoy) después se arma el html con JavaScript.