Reparto

(1) Arquitectura actual de Ember (Octane / Polaris) y módulos clave

Ediciones: Octane y Polaris

- **Octane (Ember 3.15+)** introduce el modelo moderno de Ember:
    - **Glimmer components** (`@glimmer/component`)
    - **Propiedades** `@tracked` (autotracking en vez de `computed`) (guides.emberjs.com)
- **Polaris** es la edición siguiente (en preparación), ya listada oficialmente como edición junto a Octane. (emberjs.com)
- A nivel de datos, la **Polaris Edition de Ember Data** busca simplificar el modelo mental de la API y alinearse con una nueva toolchain (warp-drive, Vite, etc.). (GitHub)

Ecosistema técnico

- **Ember CLI**: CLI oficial que da:

    - pipeline de build (Broccoli + Babel),
    - servidor dev con live-reload,
    - generadores (`ember generate`) y sistema de addons,
    - soporte integrado para testing. (ember-cli.com)
- **Ember Data**: capa de datos estándar:
    - **Store** central,
    - **Models**, **adapters** (host, URLs, headers) y **serializers** (formato JSON/API, etc.). (guides.emberjs.com)
- **Glimmer**:
    - motor de render rápido (VM + DOM incremental) usado dentro de Ember,
    - compila plantillas Handlebars a instrucciones de bajo nivel para actualizaciones eficientes. (glimmerjs.com)

¿Qué entra en el 20 % "Componentes"?

Yo acotaría esta parte teórica/técnica a los bloques que realmente usa todo el resto:

1. **Modelo de componentes Octane**
    - Glimmer components (`class X extends Component`)
    - separación JS (`.js`) + template (`.hbs`)
    - "data-down, actions-up".
2. **Reactividad con `@tracked` y `@args`**
    - cómo funciona el autotracking y cuándo usar `@tracked`. (guides.emberjs.com)
3. **Plantillas, helpers y modifiers**
    - sintaxis de templates (loops, conditionals, `{{on}}`, `{{fn}}`, helpers propios).
4. **Servicios y DI**
    - `@service` para estado global (sesión, store de UI, etc.).
5. **Patrones de componentes en aplicaciones reales**

- componentes contenedor vs. presentacionales,
- lifting state a servicios o rutas.

Con esto cubres el **núcleo del sistema de UI**, sin perderte en cosas muy exóticas.

¿Qué entra en el 60 % “Caso de estudio”?

Aquí usaría módulos más amplios (app real + ecosistema):

1. **Tooling y configuración**
   - crear proyecto con Ember CLI, estructura del repo, scripts, testing integrado. (ember-cli.com)
2. **Ruteo y navegación avanzada**
   - rutas anidadas, dynamic segments, query params, loading/error substates. (developer.mozilla.org)
3. **Capa de datos con Ember Data**
   - modelos, relaciones, adapters/serializers para una API real (RealWorld),
   - manejo de errores y estados de carga.
4. **Autenticación y autorización**
   - login/signup, gestión de sesión (p. ej. con Ember Simple Auth) y protección de rutas.
5. **CRUD completo y formularios “ricos”**
   - crear/editar/borrar entidades (artículos, comentarios, etc.),
   - validaciones, feedback de errores, UX.
6. **Gestión de estado y servicios de dominio**
   - servicios para estado compartido (user session, filtros, notificaciones).
7. **Testing + optimización/performance**
   - acceptance/component tests,
   - pequeñas optimizaciones (lazy loading, análisis de performance). (OneNine)

---

(2) Fuentes RealWorld / tutoriales avanzados para la parte práctica

Recursos que encajan muy bien con nivel 4º Ingeniería (CRUD, auth, ruteo complejo):

Recurso	Qué aporta	Cómo encaja en la práctica
Ember RealWorld example app (gothinkster/ember-realworld) (GitHub)	Implementación Ember del clon de Medium: CRUD de artículos, comentarios, perfiles, favoritos, **autenticación con JWT**, routing y patrones de UI reales.	Base del **Caso de estudio**: tomar su dominio/arquitectura como referencia (o incluso reimplementarlo parcialmente).
Ember Octane RealWorld example app (patocallaghan – ahora fusionado en el anterior) (GitHub)	Versión pensada explícitamente para Octane (Glimmer components, `@tracked`).	Referencia concreta de cómo aplicar el modelo Octane a una app compleja.
Super Rentals tutorial (Ember Guides) (guides.emberjs.com)	Tutorial oficial moderno: routing, componentes, tests, datos remotos. Menos complejo que RealWorld, pero muy didáctico.	Ideal para la **fase inicial del caso de estudio** (bloque de configuración + primeros componentes/rutas).
YoEmber – Ember.js Octane CRUD tutorial (Yoember)	Tutorial paso a paso para construir una app CRUD en Octane (incluye forms, Ember Data, etc.).	Referencia directa para diseñar los flujos CRUD (crear/editar/borrar entidades del caso de estudio).
Real World Ember (colección de apps) (GitHub)	Colección de apps Ember reales open-source para aprender patrones.	Sirve para justificar decisiones arquitectónicas y mostrar alternativas.

Subdivisión sugerida de la parte práctica (dentro del 60 % caso de estudio):

1. **Unidad P1 – Warm-up con Super Rentals (mini-tutorial guiado)**
   - Objetivo: entender routing básico, componentes y datos remotos.
2. **Unidad P2 – Dominio RealWorld: artículos, perfiles y feed**
   - Basado en Ember RealWorld.

- Rutas anidadas, paginación, modelos y relationships.

3. **Unidad P3 – Autenticación + CRUD avanzado**
   - Login, registro, guardas de ruta, creación/edición de artículos y comentarios.
4. **Unidad P4 – Testing + refactor/optimización**
   - Tests de integración/aceptación sobre los flujos principales
   - Micro-optimizaciones + uso de Ember Inspector/perf. tools.

---

(3) Comparativa técnica Ember vs React / Vue / Angular

Convención sobre configuración y ecosistema

- Ember es **claramente "opinionated"** y se basa en *convention over configuration*: estructura de proyecto fija, router integrado, CLI, testing y capa de datos oficial. (Medium)
- React es una **librería de UI**, muy flexible pero sin capa oficial de routing o datos; hay que elegir stack (React Router, Redux, etc.).
- Vue es "progressive framework": más ligero, fácil de integrar por partes, con convenciones menos estrictas que Ember. (Jelvix)
- Angular, como Ember, es un **framework completo** (CLI, router, DI, etc.), pero mucho más centrado en TypeScript, decoradores y arquitectura modular. (Kombai)

Mensaje para tu sección de ventajas/desventajas:

> Ember → menos decisiones de arquitectura, más productividad y coherencia de código a largo plazo, a costa de ser más rígido y "distinto" al mainstream React.

Curva de aprendizaje

- Varias comparativas remarcan que Ember tiene una **curva inicial más dura** por sus conceptos propios (editions, Ember CLI, Ember Data, naming conventions). (blog.risingstack.com)

- React y Vue se perciben como más **fáciles de arrancar**, porque puedes empezar "solo con componentes" y añadir el resto on-demand. (Medium)
- A cambio, una vez superada la fase inicial, Ember ofrece una **trayectoria de aprendizaje muy lineal** ("lo que se hace en apps grandes es lo mismo que en el tutorial"). (whyember.com)

Rendimiento

- El motor **Glimmer** posiciona Ember bastante bien en rendimiento de renderizado, especialmente en actualizaciones incrementales. (GitHub)
- Aun así, comparativas recientes colocan Angular y Ember como frameworks más "pesados" en bundle y tooling, frente a React/Vue/Svelte, que destacan en rendimiento bruto. (imaginarycloud.com)
- Conclusión matizable que puedes usar:
  - para **SPAs muy grandes y de vida larga**, la productividad y estabilidad de Ember (upgrade path, CLI, Data) compensan el coste extra de peso inicial; (Prerender)
  - para **micro-frontends y UIs ultra ligeras**, React/Vue/Svelte pueden ser más adecuados.

---

(4) Subdivisión del "Caso de estudio" (60 %) en bloques

Suponiendo que el trabajo total tiene un 100 %, de los cuales:

- **20 %** → sección *Componentes*,
- **60 %** → *Caso de estudio*,

propongo dividir ese **60 %** en **4 bloques**, independientes pero conectados:

1. **CS1 – Infraestructura, CLI y API (10 % del total)**
   - Crear proyecto con `ember new`.
   - Configurar Ember CLI (scripts, builds, entorno producción). (ember-cli.com)
   - Documentar la API (RealWorld) y adaptar `environment.js`, `.env`, etc.
   - Primer smoke-test y despliegue sencillo (Netlify/Render/heroku-like).

2. **CS2 – Modelado de dominio y ruteo avanzado (20 %)**
    - Modelos Ember Data: `user`, `article`, `comment`, `tag`, relaciones. (toptal.com)
    - Adapters/serializers para hablar con la API RealWorld. (guides.emberjs.com)
    - Rutas principales: feed global, feed usuario, vista de artículo, perfil, etc. (developer.mozilla.org)
    - Rutas anidadas y estados de carga/error.
3. **CS3 – Componentes de UI, formularios y UX (20 %)**
    - Glimmer components para la UI (navbar, lista de artículos, tarjeta de artículo, forms de login/signup, editor de artículo). (guides.emberjs.com)
    - Helpers y modifiers (ej. `{{on}}`, helpers de formato, componentes controlados).
    - Validación de formularios y manejo de errores de API.
4. **CS4 – Auth, estado global, testing y optimización (10 %)**
    - Integración auth (ember-simple-auth u otro patrón de sesión).
    - Servicios para estado global (usuario actual, favoritos, toasts).
    - Tests de aceptación para los flujos críticos (login + CRUD artículo). (guides.emberjs.com)
    - Breve sección de optimización: análisis de bundle, lazy routes, etc.

Estos bloques se pueden trabajar casi por separado y luego ensamblar la app final.

---

(5) Cuatro paquetes de trabajo equilibrados

Voy a definir 4 **Paquetes (P1–P4)** mezclando:

- teoría breve: Introducción, Evolución, Comparativas, Conclusiones,
- **20 % Componentes** (repartido),
- **60 % Caso de estudio** (CS1–CS4).

🔷 Paquete P1 – Intro + Arquitectura + Inicio del caso de estudio

Teoría (escrito/presentación):

- *Introducción a Ember.js*:
  - qué problema resuelve, filosofía de "framework para apps ambiciosas",
  - convención sobre configuración y ecosistema (Ember CLI, Data, Glimmer). (Medium)
- *Arquitectura general y ediciones*:
  - de "classic Ember" a **Octane**, y la futura **Polaris**. (emberjs.com)

Componentes (20 % – parte que le toca a este paquete):

- Introducir Glimmer components, estructura JS + `.hbs`, `@tracked` y `@args`. (guides.emberjs.com)

Caso de estudio (CS1 + arranque CS2):

- Crear la app base (`ember new conduit-clone`).
- Conectar con API RealWorld (config host en adapter global). (GitHub)
- Definir rutas básicas: `/`, `/login`, `/register`.
- Modelos mínimos: `user`, `article` (solo campos esenciales).
- Pintar feed básico en `/` con una lista de artículos.

🔷 Paquete P2 – Evolución + Ruteo/Modelos

Teoría:

- *Evolución de Ember y ecosistema*:
  - historia resumida (handlebars clásico → Glimmer → Octane),
  - rol de Ember CLI y Ember Data en la evolución. (ember-cli.com)

Componentes:

- Patrón **data-down, actions-up**, lifting state a rutas y servicios.
- Uso de services `@service` para compartir estado de vista (ej. filtros). (Ember.JS)

Caso de estudio (núcleo de CS2):

- Modelar completo el dominio RealWorld en Ember Data (relationships `hasMany` / `belongsTo`).

- Adapters/serializers específicos si la API no es JSON:API “perfecta”. (guides.emberjs.com)
- Rutas anidadas:
  - `/profile/:username`,
  - `article/:slug` con nested routes para comentarios.
- Manejo de estados de carga y error (substates y plantillas dedicadas).

🔷 Paquete P3 – Comparativas + Componentes/UI + CRUD

Teoría:

- *Comparativa Ember vs React/Vue/Angular*:
  - convención vs configuración,
  - curva de aprendizaje,
  - rendimiento y casos de uso típicos (Ember para productos grandes y duraderos). (blog.risingstack.com)

Componentes (parte más gorda del 20 %):

- Diseño de componentes de UI del caso de estudio:
  - Card de artículo, lista paginada, navbar, paginador, etc.
- Formularios y validación (login, signup, editor de artículo). (Yoember)
- Uso de helpers y modifiers (`{{on}}`, `{{each}}`, helpers propios).

Caso de estudio (CS3 – UI/UX + CRUD):

- Implementar:
  - Crear/editar/borrar artículo.
  - Crear/borrar comentario.
- UX “decente”: estados de error de validación, mensajes al usuario, loading en botones.

🔷 Paquete P4 – Auth + Estado global + Testing + Conclusiones/Futuro

Teoría:

- *Conclusiones y futuro de Ember*:
  - papel de Ember hoy frente a React/Vue,
  - Polaris, Vite y Ember Data 6.0 como siguiente salto. (emberjs.com)

Componentes (última parte del 20 %):

- Componentes "cross-cutting": layout principal, componente de notificaciones, etc.
- Patrón de servicios de dominio (ej. servicio `session`, `notifications`).

Caso de estudio (CS4 completo):

- Autenticación:
  - flujo de login/signup,
  - persistencia de token (localStorage o similar),
  - protección de rutas (hooks `beforeModel`, redirecciones). (programwitherik.com)
- Estado global:
  - servicio `session` con usuario actual,
  - servicio `ui-state` o `notifications`.
- Testing:
  - tests de aceptación para: login, crear artículo, editar y borrar.
  - algún test de componente clave (card de artículo, formulario).
- Mini-sección de performance:
  - medición básica,
  - una o dos optimizaciones justificadas (lazy route, split de componentes). (OneNine)
