МДК 09.01. Проектирование и разработка веб-приложений

Лекция: Современные подходы к разработке приложений (SPA, REST, CI/CD)

Цель лекции: Понять философию и практическое применение трех ключевых современных подходов, которые определяют процесс создания, функционирования и доставки веб-приложений.

План лекции:

1. **Введение:** Эволюция веба — от статических страниц к сложным приложениям.
2. **Single Page Application (SPA):** Архитектура, принципы работы, преимущества и недостатки.
3. **REST API:** Концепция единого интерфейса между клиентом и сервером.
4. **CI/CD (Continuous Integration / Continuous Deployment):** Практики автоматизации процесса разработки и поставки кода.
5. **Синтез: Как SPA, REST и CI/CD работают вместе.**
6. **Резюме и ключевые выводы.**

---

1. Введение: Эволюция веба

Раньше веб-сайты были простыми коллекциями статических HTML-страниц. Каждое действие пользователя — клик по ссылке, отправка формы — вело к полной перезагрузке страницы с сервера. Это модель **Multi-Page Application (MPA)**.

С появлением Gmail, Google Maps в середине 2000-х мир увидел новый подход: приложение, которое работает внутри браузера и динамически подгружает данные, не перезагружая страницу полностью. Это был прорыв в UX (User Experience). Этот подход стал возможным благодаря активному использованию JavaScript и получил название **Single Page Application (SPA)**.

Но для того чтобы SPA-клиент мог общаться с сервером и запрашивать данные, нужен был универсальный, стандартизированный язык. Таким языком стал **REST API**.

А чтобы быстро и надежно разрабатывать, тестировать и развертывать такие сложные приложения, состоящие из независимого фронтенда (SPA) и бэкенда (REST API), индустрия пришла к практике **CI/CD**.

Давайте разберем каждый из этих кирпичиков современной разработки.

---

2. Single Page Application (SPA)

Что это? Это веб-приложение, которое загружает единственную HTML-страницу и все необходимые ресурсы (JS, CSS). Последующие взаимодействия с пользователем динамически обрабатываются JavaScript, который запрашивает данные у сервера и обновляет интерфейс без полной перезагрузки страницы.

Ключевой принцип: Разделение ответственности. Сервер (бэкенд) отвечает только за данные и бизнес-логику. Клиент (браузер) отвечает за отображение, UI-логику и взаимодействие с

пользователем. Связь между ними происходит через API (чаще всего RESTful).

Как работает?

1. Пользователь заходит на сайт (`https://myapp.com`).
2. Сервер отдает единственный `index.html` и подключаемые к нему JS/CSS файлы (часто собранные в один бандл).
3. Загруженный JavaScript-код (например, React, Vue, Angular) инициализирует приложение, рендерит первоначальный интерфейс.
4. Приложение асинхронно (AJAX/fetch) запрашивает у бэкенда необходимые данные (например, список задач, профиль пользователя).
5. Получив данные (обычно в формате JSON), SPA обновляет только конкретные части DOM-дерева, а не всю страницу.

Пример: Лента новостей в социальной сети.

- В MPA: чтобы увидеть новые посты, вы нажимаете F5, страница полностью перезагружается.
- В SPA: вы просто листаете ленту. JavaScript-код незаметно для вас отправляет запрос на сервер (`GET /api/posts?offset=10`), получает JSON с новыми постами и плавно добавляет их вниз списка.

Преимущества:

- **Более быстрый UX:** После первоначальной загрузки приложение работает очень отзывчиво, похоже на нативное.
- **Разделение фронтенда и бэкенда:** Команды могут работать параллельно, согласовывая только формат API.
- **Богатый интерфейс:** Возможность создавать сложные, интерактивные интерфейсы.

Недостатки:

- **SEO (Search Engine Optimization):** Традиционные поисковые боты плохо индексируют JS-контент. Решается с помощью **SSR (Server-Side Rendering)** — например, Next.js для React.
- **Первоначальная загрузка:** Первый раз может грузиться дольше, так как нужно скачать весь фреймворк. Решается ленивой загрузкой, оптимизацией бандлов.
- **Требует мощного клиента:** Вся логика рендеринга лежит на JavaScript, что может нагружать слабые устройства.

---

3. REST API

Что это? REST (Representational State Transfer) — это архитектурный стиль, набор принципов для построения веб-сервисов. **API (Application Programming Interface)** — это интерфейс для взаимодействия между программами. **REST API** — это интерфейс, который следует принципам REST.

Ключевые принципы (Очень важны!):

1. **Единообразие интерфейса (Uniform Interface):** Самый важный принцип. API должен быть предсказуемым.
    - **Ресурсы:** Все данные представляются как ресурсы (например, `User`, `Post`, `Product`). Доступ к ним осуществляется через уникальный URL (например, `/api/users/123`).

    - **HTTP-глаголы (Методы):** Действия над ресурсами определяются HTTP-методами:
        - `GET` — получить ресурс (или список).
        - `POST` — создать новый ресурс.
        - `PUT` / `PATCH` — обновить существующий ресурс (полностью/частично).
        - `DELETE` — удалить ресурс.
    - **Пример для ресурса `Article`:**
        - `GET /api/articles` → Получить список всех статей.
        - `POST /api/articles` → Создать новую статью (данные в теле запроса).
        - `GET /api/articles/5` → Получить статью с id=5.
        - `PUT /api/articles/5` → Полностью обновить статью с id=5.
        - `DELETE /api/articles/5` → Удалить статью с id=5.
2. **Отсутствие состояния (Stateless):** Каждый запрос от клиента к серверу должен содержать всю информацию, необходимую для его понимания. Сервер не хранит состояние сессии клиента между запросами. (Аутентификация через токены, например, JWT, которые передаются в каждом запросе).
3. **Кэшируемость:** Ответы сервера должны явно помечаться как кэшируемые или нет. Это повышает производительность.
4. **Слоистая система:** Клиент не знает, подключен ли он напрямую к серверу или через промежуточные слои (кэш, балансировщик нагрузки). Это улучшает масштабируемость.

Пример "Нерестового" (RPC-like) подхода vs REST:

- **Плохо (не REST):** `POST /api/getUserProfile` или `POST /api/updateUserName`. Здесь URL определяет действие, а не ресурс.
- **Хорошо (REST):** `GET /api/users/123` (получить) и `PATCH /api/users/123` (обновить, передав в теле только поле `name`).

Формат данных: Чаще всего JSON (JavaScript Object Notation) как легковесный и легкочитаемый формат.

```
// Ответ сервера на GET /api/users/1
{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com"
}
```

---

4. CI/CD (Continuous Integration / Continuous Deployment)

Это культура и набор практик, которые с помощью инструментов автоматизации позволяют часто и надежно развертывать новые версии приложения.

CI (Continuous Integration — Непрерывная интеграция):

- **Цель:** Автоматически проверять, что новый код, который разработчики сливают в общую ветку (чаще `main`/`master`), не ломает существующую кодовую базу.
- **Как работает?**

1. Разработчик делает `git push` в репозиторий (например, GitHub, GitLab).
2. Запускается **CI-сервер** (например, GitHub Actions, GitLab CI, Jenkins).
3. Сервер берет свежий код и запускает заранее настроенный **пайплайн (pipeline)**:
   - Сборка проекта (`npm build`, `mvn compile`).
   - Запуск линтеров (проверка стиля кода).
   - Запуск всех модульных и интеграционных тестов.
4. Если все этапы прошли успешно (зеленый свет), код считается готовым к дальнейшим стадиям. Если нет — разработчик получает уведомление о том, что нужно исправить ошибки.

CD (Continuous Deployment / Delivery — Непрерывное развертывание / поставка):

- **Continuous Delivery:** Это расширение CI. После успешного прохода тестов код автоматически подготавливается к релизу (деплою) в среду, похожую на продакшен (staging). Развертывание на боевой сервер происходит вручную по нажатию кнопки.
- **Continuous Deployment:** Идет еще дальше. Любое изменение, успешно прошедшее CI-пайплайн, **автоматически** развертывается на боевые серверы без какого-либо ручного вмешательства. Это требует высочайшего уровня автоматизации и доверия к тестам.

Простой пример пайплайна в `.gitlab-ci.yml`:

```
stages:
  - test
  - build
  - deploy

unit_tests:
  stage: test
  script:
    - npm install
    - npm run test

build_project:
  stage: build
  script:
    - npm run build
  artifacts:
    paths:
      - build/

deploy_to_staging:
  stage: deploy
  script:
    - echo "Deploying to staging server..."
    - scp -r build/* user@staging-server:/var/www/app
  only:
    - main # Эта задача запустится только для ветки main
```

Преимущества CI/CD:

- **Скорость:** Вы можете выпускать обновления ежедневно, ежечасно.
- **Надежность:** Автоматические тесты ловят баги до того, как они попадут к пользователям.
- **Предсказуемость:** Процесс стандартизирован и повторяем.
- **Быстрая обратная связь:** Разработчик сразу узнает, если его код сломал сборку.

---

5. Синтез: Как SPA, REST и CI/CD работают вместе

Представьте себе процесс разработки современного приложения — условного "Трелло":

1. **Архитектура:** Мы решили сделать SPA на React и бэкенд на Node.js с REST API.
2. **Разработка:**
   - Фронтенд-разработчик пишет компонент для отображения доски. Ему нужны задачи. Он договаривается с бэкенд-разработчиком, что тот сделает эндпоинт `GET /api/boards/{id}/tasks`.
   - Бэкендер создает этот эндпоинт, который возвращает JSON-массив задач. Пока эндпоинт не готов, фронтендер использует мок-данные.
3. **Интеграция:**
   - Оба разработчика пушат код в одну ветку Git.
   - Срабатывает **CI/CD-пайплайн**:
     - Для бэкенда: запускаются юнит-тесты моделей, линтеры.
     - Для фронтенда: запускаются тесты компонентов, сборка проекта. Если сборка упала — значит, фронтенд-разработчик что-то сломал и получит уведомление.
   - Если все тесты прошли, пайплайн автоматически деплоит новую версию бэкенда на staging-сервер, а собранные статические файлы фронтенда — на CDN (сеть доставки контента).
4. **Работа приложения:**
   - Пользователь открывает приложение. Браузер загружает SPA с CDN.
   - SPA делает запрос `GET /api/boards/123/tasks` к бэкенду.
   - Бэкенд, работающий на сервере, обрабатывает REST-запрос, ходит в базу данных и возвращает JSON.
   - SPA получает JSON и красиво рисует задачи на доске.

Все три технологии идеально дополняют друг друга, создавая эффективный цикл разработки и отличный пользовательский опыт.

---

6. Резюме и ключевые выводы

1. **SPA** — это архитектура клиентской части, которая предоставляет богатый и отзывчивый пользовательский интерфейс, работая как динамическое приложение в браузере и общаясь с сервером через API.
2. **REST API** — это стандартный, предсказуемый и масштабируемый способ организации взаимодействия между клиентом (SPA) и сервером. Он основан на концепции ресурсов и HTTP-методов.
3. **CI/CD** — это критически важная практика DevOps для автоматизации процесса поставки ПО. Она обеспечивает высокую скорость выпуска обновлений, надежность и стабильность работы приложения, что особенно важно для сложных проектов, разделенных на фронтенд и бэкенд.

4. **Вместе** эти подходы формируют **стандарт де-факто** для разработки современных высоконагруженных, легко масштабируемых и удобных в поддержке веб-приложений. Понимание их взаимодействия — ключевой навык для любого веб-разработчика.